IN-62
36556

NASA Contractor Report 187574

Verification of the FtCayuga Fault-Tolerant Microprocessor System

Volume 2: Formal Specification and Correctness Theorems

Mark Bickford and Mandayam Srivas

ORA Corporation
Ithaca, New York

Contract NAS1-18972
July 1991

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Abstract

This is the second volume of a two-volume report that presents a formal specification and verification of a property of a quadruplicately redundant fault-tolerant microprocessor system design. This volume gives a complete listing of the formal specification of the system and the correctness theorems that were proved. The system performs the task of attaining *interactive consistency* among the processors using a special instruction on the processors. The design is based on an algorithm proposed by Pease, Shostak and Lamport. The microprocessor used in the system is called FtCayuga, which was designed by extending another formally verified microprocessor MiniCayuga. The property verified ensures that an execution of the special instruction by the processors correctly accomplishes interactive consistency, provided certain preconditions hold, using a computer-aided hardware design verification tool, Spectool, and the theorem prover, Clio, both of which were developed at ORA. A major contribution of the work is the demonstration of a significant fault-tolerant hardware design that is mechanically verified by a theorem prover.

Key Words: Fault-tolerance, hardware verification, mechanical theorem proving.

Contents

1 Introduction

NASA Langley Research Center has recently initiated a research effort to develop a validation methodology for life-critical digital (fly-by-wire) flight control systems. Such systems must meet stringent reliability requirements. The systems are expected to have a probability of failure as low as 10^{-9} for a 10 hour mission. Hence, as has been well-argued in [6], the design and validation methods employed for such systems must meet high standards. The designs must use fault-tolerant strategies to enable the continued operation of the system in the presence of component failures. The validation methods must ensure that there are no design errors in the system.

The process of formal verification is capable of showing that a system design satisfies a specified property for all inputs and initial conditions of the design. Hence it is a promising candidate for consideration as a validation technology for flight control systems. Digital control systems are implemented using a combination of hardware and software components. Hence, a formal verification technology for flight control systems must be applicable to both software and hardware designs. This work is concerned with the formal verification of a hardware design.

This report presents the formal verification of a hardware system for a task that is an important component of a fault-tolerant computer architecture for flight control systems. The hardware system implements an algorithm for attaining *interactive consistency* (*byzantine agreement*) [2] among four microprocessors as a special instruction on the processors. The property verified ensures that an execution of the special instruction by the processors correctly accomplishes interactive consistency, provided certain preconditions hold. An assumption is made that the processors execute synchronously. For verification, we used a computer-aided hardware design verification tool, Spectool [3], and the theorem prover, Clio [1], both of which were developed at ORA. The microprocessor used in the system is called FtCayuga, which we designed by extending the the formally verified microprocessor MiniCayuga [4].

A major contribution of the work is the demonstration of a significant fault-tolerant hardware design that is mechanically verified by a theorem prover. The work illustrates the advantage of using hierarchy and abstraction in system design specification to manage the complexity of formal verification. The work demonstrates the value of a special-purpose tool that tailors the use of a theorem prover to a hardware domain in order to reduce the effort

required for formal verification.

This is the second volume of a two-volume report. The first volume [5] contains an overview of the work and an informal description of the specification and the proof of correctness. This volume contains the entire specification of the system and the definitions of all the correctness theorems and lemmas we proved. This volume is intended as a supplement to Volume 1, and, hence, the presentation in this volume assumes that you have read Volume 1.

The next section describes the general structure of a specification generated by Spectool, which also appeared as appendix section A in Volume 1. We repeat this description here for easy reference. The description is useful in understanding the design specifications of the main blocks—`IcNet`, `Voter`, and `FtCayuga`—of our system. The rest of the material in this volume is organized as follows.

1. The Voter Specification section contains the design and abstract specifications of the Voter.

2. The FtCayuga Specification section contains the design and abstract specifications of FtCayuga.

3. The IcNet Specification section contains the design and abstract specifications of IcNet.

4. The Common Theory section contains definitions of the data types and primitive functions on the data types that are used in the previous specifications. Some of these types are unimplemented abstract types which are specified by axiomatizing the operations on them.

5. The Main Lemmas section contains the definitions of the main theorem and the main lemmas with the help of which the main theorem was proved. The bridge theorems that connect the two levels of descriptions of `IcNet` and `Voter` appear as part of the abstraction specifications of the respective components. Since we did not prove the bridge theorems for `FtCayuga`, we have not included them here.

6. The General Lemmas section contains the set of basic lemmas that were used in proving one or more of the lemmas in the Main Lemmas section.

2 General Structure of a Specification Generated by Spectool

A design specification generated by Spectool consists of the parts described in the following sections.

2.1 Structural Specification Part

This part specifies the data path architecture. It defines a set of types to model the data path state and a set of functions to specify the connections between the data path components. Spectool generates this part in a generic fashion from the following structural information about the data path provided by the user:

1. the names of components and component classes,
2. the names of the actions defined on components,
3. the types that model the internal states of components,
4. the types of the external inputs to the circuit, and
5. the graphical connections between components.

To give the reader an idea of this part of a design specification, a fragment of the structural specification part of the voter circuit is shown below.

```
|| Generic part of the data path state definition
type SYSTEM_STATE = COMP->LOCAL_STATE
type INPUT_STREAM = NAT->EXTSTATE
type CHANGE = <<COMP,NAT,LOCAL_STATE>>
type STATE = <<SYSTEM_STATE, [CHANGE], INPUT_STREAM>>

|| A type for every component class. The type defines the names of
|| all the components used in the data path that belong to the class.
majority ::= MAJ1 | MAJ2 | MAJ3
bytereg  ::= PR1 | PR2 | PR3 | R12 | R13 | R23 | R31 | R32
```

```
bitlatch ::= VS | ST

|| A labeled union type of all the component class types
COMP ::=   Controller | C_majority !majority | C_mux4 !mux4
         | C_bytereg !bytereg | C_bitlatch !bitlatch

|| A labeled union type of all local states of components
LOCAL_STATE ::=   S_Controller !CONTROLSTATE
                | S_majority !majority_localstate
                | S_mux4 !mux4_localstate
                | S_bytereg !bytereg_localstate
                | S_bitlatch !bitlatch_localstate

ACTION ::=   A_majority !majority_ACT
           | A_mux4 !mux4_ACT
           | A_bytereg !bytereg_ACT
           | A_bitlatch !bitlatch_ACT

|| The following specifies the input connections to the component MAJ2
majorityinput s (C_majority MAJ2) =
   <<getbyteregout0 (current s (C_bytereg R32)),
     getbyteregout0 (current s (C_bytereg R12)),
     getbyteregout0 (current s (C_bytereg PR2))>>
....
```

The type STATE defines the data path state as a tuple: <<SYSTEM_STATE, [CHANGE], INPUT_STREAM>>. SYSTEM_STATE maps every component (an element of type COMP) in the data path to its LOCAL_STATE. [CHANGE] is the list of *pending* actions on the data path components. This list is maintained to simulate the effect of delays on actions. The list contains an update record for each of the actions that has been triggered by the controller on components, but is yet to be completed. The first two fields of STATE define a snapshot in time of the data path state. The state of a component is given by SYSTEM_STATE unless an action is pending on the component, in which case the state is bottom.

The third field of the tuple, INPUT_STREAM, specifies the values of the external inputs to the circuit as function of time. INPUT_STREAM is a function type from time (NAT) to EXTSTATE, where EXTSTATE is a type that combines all the external inputs into a tuple.

The type LOCAL_STATE is a labeled union of the *local states* of all the components of the data path (and the controller). The local state type of a component is a tuple of its internal state and (a tuple of) its outputs. The definition of the local state type of a component appears as part of the component class specification.

The set of components in a data path is organized so that every component is an instance of a component class. The type COMP groups together the names of all the components in a data path. It is defined as a labeled union of all the component class types, where a component class type is an enumerated type of the names of all the components belonging to the corresponding class.

Similarly, the type ACTION groups together the names of the It is defined as a labeled union of all the *action types* of the component classes, where the action type of a component class is an enumerated type of the names of all the actions defined for the class. The definition of the action type of a component appears as part of the component class specification.

Connections between data path components are specified by defining, for every component, a function that determines the inputs to the component in a given data path state.

2.2 Component Classes Specification Part

Every component in the data path is an instance of a component class. The components belonging to the same class share several attributes. This part specifies the shared attributes of a class for every class used in a design. A component class specification defines the type that denotes the internal state of a components of the class, the types of the outputs, a type denoting the names of the actions on the components, and the effects of the actions on the components. For every action, it defines three functions: "state" and "output" functions return, respectively, the new state and the new outputs of a component after an action is performed; and the "delay" function gives the delay associated with the action. For illustration, a part of the specification of the majority component class used in the voter circuit is given below.

```
|| The component class majority.
|| Local state of majority: <<internal state type, <<output types>>>>
type majority_localstate = <<<<data, BOOL>>,<<(byte),(BOOL)>>>>
```

```
|| Type denoting the actions on the components of the class
majority_ACT ::=
 select3 !majority |
 select2 !majority |
 select0 !majority |
 select1 !majority |
 comp !majority

majoritydelay (select3 c) = #0
majoritycomp (select3 c) = C_majority c
majorityout (select3 c) s <<in1,in2,in3>> = <<get_byte 3 (dataof s), bitof s>>
majoritystate (select3 c) s <<in1,in2,in3>> = s

majoritydelay (select2 c) = #0
majoritycomp (select2 c) = C_majority c
majorityout (select2 c) s <<in1,in2,in3>> = <<get_byte 2 (dataof s), bitof s>>
majoritystate (select2 c) s <<in1,in2,in3>> = s
....
```

2.3 Controller Specification Part

A controller is specified by means of two functions **nextstate** and **scheduler**. The **nextstate** function gives the next controller state as a function of the current state and controller inputs. The **nextstate** function is used at the end of each cycle to advance the controller state. The function **scheduler** returns a list of actions as a function of the controller state, controller inputs, and the phase. For illustration, a part of the specification of the controller for the voter circuit is shown below.

```
CONTROLSTATE ::=  LDP1 | LDP2 | XNG11 | XNG12 | XNG21 | XNG22 | XNG31
                | XNG32 | CMPP | OUT1 | OUT2

nextstate LDP1 in = (~(startedof in))->(LDP1);(LDP2)
nextstate LDP2 in = XNG11
...

scheduler XNG11 <<vstart, started>> 0 = (selprvt1byte0)
```

```
selprvt1byte0 = [A_bytereg(read0 PR1), A_mux4(choose3 MUX)]

scheduler XNG12 <<vstart, started>> 0 = (selprvt1byte2)
selprvt1byte2 = [A_bytereg(read2 PR1), A_mux4(choose3 MUX)]
...
|| Some of the details specific to the voter design
|| There are 4 clock phases per cycle.
num_phases = 4
```

2.4 Composite Behavior Specification Part

This part defines a set of functions that derive the composite behavior of a design using the information expressed in the rest of the specification. This part formalizes in Caliban the operational model of the behavior of a finite state controller system. This part is identical for every circuit since Spectool generates it in a completely generic fashion. The higher-order function definition capability of Caliban is a primary reason why it is possible to express this part in a generic fashion. A top level fragment of this part of the specification is shown below.

```
Execute s = do_phases 0 s

Output s = generate_output 0 s

do_phases n s = update_state s , n = num_phases
                do_phases (n+1) (do_phase n s)

do_phase n <<s,p,in>> =
  advance_inputstream (do_actions (current_schedule s2 n) s2)
  where s2 = update_state <<s,p,in>>
```

The two main functions defined in this part are `Execute` and `Output`. `Execute` advances the state of the system across a single cycle. `Output` returns the (tuple of) external outputs produced by the circuit over the next cycle. `Output` actually returns a list outputs, one for every phase in a cycle. The two functions are defined hierarchically in terms of several other functions, some of which are described below.

To simulate the effect of delays on component actions, the specification maintains a list of records of pending updates that have been triggered by the controller, but are yet to take effect on the circuit state. The update record for an action contains the component on which the update is pending, a time-out counter that is initialized to the delay of the action and decremented after every phase, and the result of the update. The function `update_state` causes the result of all the update records on the list that have timed out to take effect on the circuit state. The function also decrements the time-out counter of the update records that have not yet timed-out by one unit of time. The purposes of the rest of the functions should be apparent from the names of the functions. They are summarized below.

- The function `do_phases` updates the state for all the phases in a cycle.
- The function `do_phase` updates the state for a single phase. It causes (using `update_state`) the pending updates that have timed out to take effect; gets the `current_schedule` from the controller, makes new update records (using `do_actions`) for all the actions in the schedule, and then advances the input stream by a time unit.

3 Voter Specification

3.1 Abstract Specification

```
FROM CommonTheorySec IMPORT update_byte,DATUM, Word, data, get_byte, good
FROM VoterDesignSec IMPORT controllerstate, majority_of, majority_exists,
                           OUT1,OUT2,XNG11,XNG12,XNG21,XNG22, XNG31,XNG32,
                           CMPP,LDP1,LDP2,
                           Execute, inlist_from, Proper_state, Output,
                           CONTROLSTATE, startedof, nextstate
                           get_PR1_state, get_PR2_state, get_PR3_state,
                           get_R21_state, get_R31_state,
                           get_R12_state, get_R32_state,
                           get_R13_state, get_R23_state,
                           get_MAJ1_state, get_MAJ2_state, get_MAJ3_state,
                           get_VS_state,
                           external_input_prvt1, external_input_prvt2,
                           external_input_prvt3, external_input_vin1,
                           external_input_vin2, external_input_vin3,
                           external_input_vstart, from_proc, from_vtrs,
                           map

select Zero (a:x) = a
select (Succ n) (a:x) = select n x
take Zero x = []
take (Succ n) (a:x) = a : take n x
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| Abstraction of the voterstate:

VoterABS s = <<controllerstate s, Voter_array s, Voter_maj s, goahead s>>
```

```
Voter_array s 1 1 = get_PR1_state s
Voter_array s 1 2 = get_PR2_state s
Voter_array s 1 3 = get_PR3_state s

Voter_array s 2 1 = get_R21_state s
Voter_array s 2 2 = get_R12_state s
Voter_array s 2 3 = get_R13_state s

Voter_array s 3 1 = get_R31_state s
Voter_array s 3 2 = get_R32_state s
Voter_array s 3 3 = get_R23_state s

Voter_maj s =
    <<get_MAJ1_state s, get_MAJ2_state s, get_MAJ3_state s>>

goahead s = get_VS_state s


Voter_from_proc s = map from_proc (take #4 (inlist_from s))
Voter_from_voters s = map from_vtrs (take #4 (inlist_from s))
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

HALFWORD_KIND ::= LO | HI

VoterStep <<cstate,array,maj,go>>  from_proc  from_voters=
    <<nextstate cstate <<go,go>>, newarray, newmaj, newgo>>
        where
            newarray {(cstate = LDP1) & go} = update_row1 LO array from_proc
            newarray {cstate = LDP2} = update_row1 HI array from_proc
            newarray {cstate = XNG11} = update_diagonal 0 LO array from_voters
            newarray {cstate = XNG12} = update_diagonal 0 HI array from_voters
            newarray {cstate = XNG21} = update_diagonal 1 LO array from_voters
            newarray {cstate = XNG22} = update_diagonal 1 HI array from_voters
            newarray {cstate = XNG31} = update_diagonal 2 LO array from_voters
            newarray {cstate = XNG32} = update_diagonal 2 HI array from_voters
            newarray  = array  || otherwise it's unchanged

            newmaj = (cstate = CMPP)->(maj_vector array) ; maj

            newgo = beginof (select (#2) from_proc) , cstate = OUT2
                    beginof (select (#2) from_proc) , (cstate = LDP1) & (~go)
                    go

update_row1 LO array in =
       update_array (update_array array row1 0 (proc_in (select Zero in)))
                                          row1 1 (proc_in (select (#2) in))
update_row1 HI array in =
       update_array (update_array array row1 2 (proc_in (select Zero in)))
                                          row1 3 (proc_in (select (#2) in))

update_diagonal n LO array in =
    update_array
     (update_array array (diagonal (#n)) 0 (cross_in n (select (#1) in)))
                         (diagonal (#n)) 1 (cross_in n (select (#3) in))
update_diagonal n HI array in =
    update_array
     (update_array array (diagonal (#n)) 2 (cross_in n (select (#1) in)))
                         (diagonal (#n)) 3 (cross_in n (select (#3) in))

update_array_byte array <<i,j>> b v i j = update_byte b (array i j) v
update_array_byte array <<i,j>> b v k l = array k l

update_array array [] b [] = array
update_array array (a:x) b (c:y) =
    update_array (update_array_byte array a b c) x b y
```

```
iterate Zero f x = x
iterate (Succ n) f x = iterate n f (f x)

succ 1 = 2
succ 2 = 3
succ 3 = 1
diagonal n =
     [<<2,iterate n succ 2>>,<<3,iterate n succ 3>>]
row1 = [<<1,1>>,<<1,2>>,<<1,3>>]

cross_in 2 <<v1,v2,v3>> = [v2,v1]
cross_in 1 <<v1,v2,v3>> = [v3,v1]
cross_in 0 <<v1,v2,v3>> = [v3,v2]

proc_in <<x,y,z,go>> = [x,y,z]

beginof <<x,y,z,go>> = go

compute_majority x y z   = <<majority_of x y z, majority_exists x y z>>

maj_vector array =
     << compute_majority (array 1 1) (array 2 1) (array 3 1),
       compute_majority (array 1 2) (array 2 2) (array 3 2),
       compute_majority (array 1 3) (array 2 3) (array 3 3)>>


||**********************************************************************
|| Corespondence between the levels:

Voter_Step_lemma :=
    Proper_state 's'
     => 'VoterABS (Execute s)'
         ='VoterStep (VoterABS s)(Voter_from_proc s)(Voter_from_voters s)'
||**********************************************************************
|| Voter output functions.  We define one function for every output
|| of the Voter.  The function defines the list of output signals
|| produced at that port in the "output phases" (0 and 2) of a cycle.


maj_out n <<cstate,array,maj,go>>
                    = low_half (get_maj_val n maj) , (cstate = OUT1)
                      high_half (get_maj_val n maj) , (cstate = OUT2)
                      dont_care || don't care in other cases

dont_care = <<bottom,bottom>>

get_maj_val 1 <<<<b0,b1>>,<<c0,c1>>,<<d0,d1>>>> = b0
get_maj_val 2 <<<<b0,b1>>,<<c0,c1>>,<<d0,d1>>>> = c0
get_maj_val 3 <<<<b0,b1>>,<<c0,c1>>,<<d0,d1>>>> = d0

low_half (DATUM (Word byte3 byte2 byte1 byte0)) = <<byte0, byte1>>
high_half (DATUM (Word byte3 byte2 byte1 byte0)) = <<byte2, byte3>>

crossout <<cstate,array,maj,go>>
    = xng_out cstate array   || depends only on cstate and array

xng_out cs array
               = sendlow 1 array, (cs = XNG11)
                 sendhigh 1 array, (cs = XNG12)
                 sendlow 2 array, (cs = XNG21)
                 sendhigh 2 array, (cs = XNG22)
                 sendlow 3 array, (cs = XNG31)
                 sendhigh 3 array, (cs = XNG32)
                 dont_care  || don't care, otherwise

sendlow n array  = <<get_byte 0 (array 1 n), get_byte 1 (array 1 n)>>
```

```
sendhigh n array = <<get_byte 2 (array 1 n), get_byte 3 (array 1 n)>>

statusout <<cstate,array,maj, go>> = getstatus maj, (cstate ~= CMPP)
                                     dont_care

getstatus <<<<maj11,maj12>>,<<maj21,maj22>>,<<maj31,maj32>>>>
                    = <<<<maj12,maj22,maj32>>,<<maj12,maj22,maj32>>>>

sndngout <<cstate,array,maj,go>>
            = <<True,True>>, (cstate = OUT1)
              <<True,True>>, (cstate = OUT2)
              <<False,False>>

freeout <<cstate,array,maj,go>>
            = <<True,True>>, (cstate = LDP1)
              <<False,False>>

|| Now the tuple <<maj_out 1 s, maj_out 2 s, ...., freeout s>> will be a tuple
|| of lists of length two. The output function at the lower level
|| gives a list of tuples of lenght two. To compare the two levels,
|| we must "transpose" one of the representations.


transpose [ <<a0,b0,c0,d0,e0,f0,g0>>, <<a1,b1,c1,d1,e1,f1,g1>>]
 = << <<a0,a1>>,<<b0,b1>>,<<c0,c1>>,<<d0,d1>>,<<e0,e1>>,
      <<f0,f1>>,<<g0,g1>> >>

VoterOut s = <<maj_out 1 s,maj_out 2 s,maj_out 3 s,statusout s,
               crossout s,sndngout s,freeout s>>

VoterOut_Lemma := 'VoterOut (VoterABS s)'
                      <= 'transpose (Output s)', Proper_state 's'

|| Now we will write another description of the voter, similar to the one
|| above, but with the inputs and outputs grouped differently and with
|| the input stream separated from the state. This description is used
|| when the voter is thought of as a component in a larger system.

type ARRAY = NUM->NUM->data
type MAJSTATE = <<data,BOOL>>
type MAJROW = <<MAJSTATE,MAJSTATE,MAJSTATE>>
type VOTERSTATE = <<CONTROLSTATE,ARRAY,MAJROW,BOOL>>

|| The input tuple used above has the form <<pvt1,pvt2,pvt3,v1,v2,v3,b>>,
|| and each cycle consumes four such tuples from the input stream.
|| The new description of the voter is as a component having
|| four inputs from_proc = <<go,pvt1,pvt2,pvt3>> , and v1, v2, and v3,
|| each of which is a four tuple (representing the values on the input wires
|| in each of the four phases).

serial_from_proc  <<<<<<go0,go1,go2,go3>>,<<pvt10,pvt11,pvt12,pvt13>>,
                    <<pvt20,pvt21,pvt22,pvt23>>,
                    <<pvt30,pvt31,pvt32,pvt33>>>>,v1,v2,v3>> =
    [<<pvt10,pvt20,pvt30,go0>>,
     <<pvt11,pvt21,pvt31,go1>>,
     <<pvt12,pvt22,pvt32,go2>>,
     <<pvt12,pvt22,pvt32,go2>>]

serial_from_voter <<pr,<<v10,v11,v12,v13>>,<<v20,v21,v22,v23>>,
                    <<v30,v31,v32,v33>>>> =
    [<<v10,v20,v30>>,
     <<v11,v21,v31>>,
     <<v12,v22,v32>>,
     <<v13,v23,v33>>]

|| So, in the new description, the voter has a state s :: VOTERSTATE
|| and it gets an input in = <<from_proc, from_v1, from_v2, from_v3>>.
```

```
|| We can define its new state by using the function VoterStep defined above.

voterstep s in = VoterStep s (serial_from_proc in)(serial_from_voter in)

|| The voter-component will have four output ports: to_proc, cross1, cross2,
|| and cross3. The "cross" outputs will all be the same:

cross_out <<cs,array,mj,b>> = pad (xng_out cs array)

pad <<a,b>> = <<a,a,b,b>>

|| The output "to_proc" contains all the other output of the voter:


to_proc s = drop_cross (VoterOut s)

drop_cross <<m1,m2,m3,status,cross,sndng,free>> =
                                        <<m1,m2,m3,status,sndng,free>>


good_array a =
    (good (a 1 1)) & (good (a 1 2)) & (good (a 1 3))
    & (good (a 2 1)) & (good (a 2 2)) & (good (a 2 3))
    & (good (a 3 1)) & (good (a 3 2)) & (good (a 3 3))

good_maj_row <<a,b,c>> = good_maj a & (good_maj b) & (good_maj c)
good_maj <<a,b>> = good a & !b

||clio: sy good_array ex
||clio: sy good_maj_row ex
||clio: sy good_maj ex

Proper_voterstate `<<cs,array,mj,go>>` :=
`!cs`=`True`
& `good_array array`=`True`
& `good_maj_row mj`=`True`
& `!go`=`True`

||*********************************************************************

Proper_VoterABS_lemma :==
     Proper_state `s` => Proper_voterstate `VoterABS s`

||*********************************************************************
|| Here are some functions we need to specify the next level of abstraction.

control <<cs,array,maj,go>> = cs
arrayof <<cs,array,maj,go>> = array
maj_of <<cs,array,maj,go>> = maj
go_of <<cs,array,maj,go>> = go
row1of <<cs,array,maj,go>> = <<array 1 1, array 1 2, array 1 3>>

|| Now we'll define a predicate that says how well defined the inputs
|| to the voter need to be in order to result in a Proper_voterstate.
good_voter_in cs in = !!(serial_from_voter in) , xng_cycle cs
                      !!(serial_from_proc in)

xng_cycle XNG11 = True
xng_cycle XNG12 = True
xng_cycle XNG21 = True
xng_cycle XNG22 = True
xng_cycle XNG31 = True
xng_cycle XNG32 = True
xng_cycle cs = False

Proper_voterstate_lemma :=
     (Proper_voterstate `s` & `good_voter_in (control s) in`=`True`)
                          => Proper_voterstate `voterstep s in`
```

```
|| When the voter state is not LDP1 the the nextstate is a function
|| of the current state only. We need to export this function:
next_voterstate s = nextstate s bottom
```

3.2 Design Specification

```
FROM CommonTheorySec IMPORT data, byte, good, BYTE0, BYTE1, BYTE2, BYTE3,
                            set_byte, byte_inc, get_byte, update_byte

good_mstate <<s,b>> c = good s & !b & (b=c)
||clio: sy good_mstate extend


|| Generated by the spectool
||clio: symbol Execute never
||clio: mod * off
||clio: add *
||clio: mod * on
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The Controller.

COMP ::= Controller |+
ACTION ::= Advance_controller|+
CONTROLSTATE :: type
LOCAL_STATE ::= S_Controller !CONTROLSTATE |+
getcontrolstate (S_Controller x) = x
is_proper_contr (S_Controller x) = !x
||clio: sy is_proper_contr extend
controllerstate s = getcontrolstate (current s Controller)
effect Advance_controller s c =
     S_Controller (nextstate (controllerstate s) (controllerinput s))
delay Advance_controller = #0
component Advance_controller = Controller

controllerinput s = <<getbitlatchout0 (current s (C_bitlatch VS)),
                      getbitlatchout0 (current s (C_bitlatch ST))>>
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The Control states and the next_state function.

vstartof <<vstart, started>> = vstart
startedof <<vstart, started>> = started

CONTROLSTATE ::= LDP1 |+
nextstate LDP1 in = (~(startedof in))->(LDP1);(LDP2)
CONTROLSTATE ::= LDP2 |+
nextstate LDP2 in = XNG11
CONTROLSTATE ::= XNG11 |+
nextstate XNG11 in = XNG12
CONTROLSTATE ::= XNG12 |+
nextstate XNG12 in = XNG21
CONTROLSTATE ::= XNG21 |+
nextstate XNG21 in = XNG22
CONTROLSTATE ::= XNG22 |+
nextstate XNG22 in = XNG31
CONTROLSTATE ::= XNG31 |+
nextstate XNG31 in = XNG32
CONTROLSTATE ::= XNG32 |+
nextstate XNG32 in = CMPP
CONTROLSTATE ::= CMPP |+
nextstate CMPP in = OUT1
```

```
CONTROLSTATE ::= OUT1 |+
nextstate OUT1 in = OUT2
CONTROLSTATE ::= OUT2 |+
nextstate OUT2 in = LDP1
CONTROLSTATE ::= |
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The component External.

type EXTSTATE = <<byte, byte, byte, byte, byte, byte, BOOL>>

external_input_prvt1 <<x0, x1, x2, x3, x4, x5, x6>> = x0
external_input_prvt2 <<x0, x1, x2, x3, x4, x5, x6>> = x1
external_input_prvt3 <<x0, x1, x2, x3, x4, x5, x6>> = x2
external_input_vin1 <<x0, x1, x2, x3, x4, x5, x6>> = x3
external_input_vin2 <<x0, x1, x2, x3, x4, x5, x6>> = x4
external_input_vin3 <<x0, x1, x2, x3, x4, x5, x6>> = x5
external_input_vstart <<x0, x1, x2, x3, x4, x5, x6>> = x6
is_proper_ext <<x0, x1, x2, x3, x4, x5, x6>> =
    (!x0) & (!x1) & (!x2) & (!x3) & (!x4) & (!x5) & (!x6)
||clio: sy is_proper_ext extend

current_input <<s,p,in>> = in Zero
nth_input n <<s,p,in>> = in n
||clio: sy Proper_External extend
Proper_External `<<s,p,in>>` :=
(t::NAT) `is_proper_ext (in t)`='True`
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| There are 4 clock phases per cycle.

num_phases = 4
input_phases  = [#0, #1, #2, #3]

output_phase 0 = True
output_phase 2 = True
output_phase n = False

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The generic Execute function.

type SYSTEM_STATE = COMP->LOCAL_STATE
type INPUT_STREAM = NAT->EXTSTATE
type CHANGE = <<COMP,NAT,LOCAL_STATE>>
type STATE = <<SYSTEM_STATE, [CHANGE], INPUT_STREAM>>

pending_changes <<s,p,in>> = p

current <<s,p,in>> c = bottom, pending p c
                       s c

pending [] c = False
pending (<<c,t,v>>:rest) c = True
pending (<<c2,t,v>>:rest) c = pending rest c

Execute s = do_phases 0 s


do_phases n s = update_state s , n = num_phases
                do_phases (n+1) (do_phase n s)

Output s = generate_output 0 s


generate_output n s {n=num_phases} = []
generate_output n s {output_phase n} =
                   Out(do_phase n s) : generate_output (n+1) (do_phase n s)
```

```
generate_output n s = generate_output (n+1) (do_phase n s)

||clio: modify_rule "do_phases" count 20000
||clio: symbol do_phases never

update_state <<s,p,in>> = do_changes p <<s,[],in>>

do_phase n <<s,p,in>> =
  advance_inputstream (do_actions (current_schedule s2 n) s2)
  where s2 = update_state <<s,p,in>>

current_schedule s n = scheduler (controllerstate s) (controllerinput s) n

do_change <<c,Zero,v>> <<s,p,in>> = <<update s c v, p, in>>
do_change <<c,Succ n,v>> <<s,p,in>> = <<s, <<c,n,v>>:p, in>>

do_action a s = do_change (change_of a s) s

advance_inputstream <<s,p,in>> = <<s,p,in.Succ>>

change_of a s = <<component a, delay a, effect a s (component a)>>

do_changes [] s = s
do_changes (<<c,t,v>>:rest) s = do_changes rest (do_change <<c,t,v>> s)
|| This is a trick to handle conditional actions correctly
do_actions :: [ACTION]->STATE->STATE
AXIOM (s)'do_actions [] s' = 's'
AXIOM (a)(rest)(s)'do_actions (a:rest) s' = 'do_actions rest (do_action a s)'
update s c v c2 = (c=c2)->v; s c2

foldl op s [] = s
foldl op s (a:rest) = foldl op (op a s) rest

map f [] = []
map f (a:x) = (f a): (map f x)
list_to Zero = []
list_to (Succ n) = list_to n ++ [n]
iterate f Zero s = s
iterate f (Succ n) s = iterate f n (f s)
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Component Classes.

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class majority.

majority :: type
type majority_localstate = <<<<data, BOOL>>,<<(byte),(BOOL)>>>>
LOCAL_STATE ::= S_majority !majority_localstate |+
ACTION ::= A_majority !majority_ACT |+
majority_ACT ::=
 select3 !majority |
 select2 !majority |
 select0 !majority |
 select1 !majority |
 comp !majority

majoritydelay (select3 c) = #0
majoritycomp (select3 c) = C_majority c
majorityout (select3 c) s <<in1,in2,in3>> = <<get_byte 3 (dataof s), bitof s>>
majoritystate (select3 c) s <<in1,in2,in3>> = s

majoritydelay (select2 c) = #0
majoritycomp (select2 c) = C_majority c
majorityout (select2 c) s <<in1,in2,in3>> = <<get_byte 2 (dataof s), bitof s>>
majoritystate (select2 c) s <<in1,in2,in3>> = s
```

```
majoritydelay (select0 c) = #0
majoritycomp (select0 c) = C_majority c
majorityout (select0 c) s <<in1,in2,in3>> = <<get_byte 0 (dataof s), bitof s>>
majoritystate (select0 c) s <<in1,in2,in3>> = s

majoritydelay (select1 c) = #0
majoritycomp (select1 c) = C_majority c
majorityout (select1 c) s <<in1,in2,in3>> = <<get_byte 1 (dataof s), bitof s>>
majoritystate (select1 c) s <<in1,in2,in3>> = s

majoritydelay (comp c) = #2
majoritycomp (comp c) = C_majority c
majorityout (comp c) s <<in1,in2,in3>> =
        <<get_byte 0 (majority_of in1 in2 in3), majority_exists in1 in2 in3>>
majoritystate (comp c) s <<in1,in2,in3>> =
        <<majority_of in1 in2 in3, majority_exists in1 in2 in3>>

effect (A_majority a) = majorityeffect a
delay (A_majority a) = majoritydelay a
component (A_majority a) = majoritycomp a
majorityeffect a s c =
   S_majority <<majoritystate a (getmajoritystate (current s c))
                                                (majorityinput s c),
               majorityout a (getmajoritystate (current s c))
                                                   (majorityinput s c)>>
getmajoritystate (S_majority <<x,y>>) = x
getmajorityout0 (S_majority <<x,<<y0,y1>> >>)  = y0
getmajorityout1 (S_majority <<x,<<y0,y1>> >>)  = y1
is_proper_majority (S_majority <<x,<<y0,y1>> >>) = (! y0) & (! y1)
||clio: sy is_proper_majority extend
Goodmajority 's' 'out' 'fnd':== 'good_mstate s fnd'='True'

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class mux4.

mux4 :: type
type mux4_localstate = (byte)
LOCAL_STATE ::= S_mux4 !mux4_localstate |+
ACTION ::= A_mux4 !mux4_ACT |+
mux4_ACT ::=
 choose4 !mux4 |
 choose3 !mux4 |
 choose2 !mux4 |
 choose1 !mux4

mux4delay (choose4 c) = #0
mux4comp (choose4 c) = C_mux4 c
mux4out (choose4 c) <<in1,in2,in3,in4>> = in4

mux4delay (choose3 c) = #0
mux4comp (choose3 c) = C_mux4 c
mux4out (choose3 c) <<in1,in2,in3,in4>> = in3

mux4delay (choose2 c) = #0
mux4comp (choose2 c) = C_mux4 c
mux4out (choose2 c) <<in1,in2,in3,in4>> = in2

mux4delay (choose1 c) = #0
mux4comp (choose1 c) = C_mux4 c
mux4out (choose1 c) <<in1,in2,in3,in4>> = in1

effect (A_mux4 a) = mux4effect a
delay (A_mux4 a) = mux4delay a
component (A_mux4 a) = mux4comp a
```

```
mux4effect a s c = S_mux4 (mux4out a (mux4input s c))
getmux4out0 (S_mux4 y) = y
is_proper_mux4 (S_mux4 y) = ! y
||clio: sy is_proper_mux4 extend

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class bytereg.

bytereg :: type
type bytereg_localstate = <<data,<<(data),(byte)>>>>
LOCAL_STATE ::= S_bytereg !bytereg_localstate |+
ACTION ::= A_bytereg !bytereg_ACT |+
bytereg_ACT ::=
 read3 !bytereg |
 read2 !bytereg |
 read1 !bytereg |
 read0 !bytereg |
 set3 !bytereg |
 set2 !bytereg |
 set1 !bytereg |
 set0 !bytereg

byteregdelay (read3 c) = #0
byteregcomp (read3 c) = C_bytereg c
byteregout (read3 c) s din = <<s, get_byte 3 s>>
byteregstate (read3 c) s din = s

byteregdelay (read2 c) = #0
byteregcomp (read2 c) = C_bytereg c
byteregout (read2 c) s din = <<s, get_byte 2 s>>
byteregstate (read2 c) s din = s

byteregdelay (read1 c) = #0
byteregcomp (read1 c) = C_bytereg c
byteregout (read1 c) s din = <<s, get_byte 1 s>>
byteregstate (read1 c) s din = s

byteregdelay (read0 c) = #0
byteregcomp (read0 c) = C_bytereg c
byteregout (read0 c) s din = <<s, get_byte 0 s>>
byteregstate (read0 c) s din = s

byteregdelay (set3 c) = #1
byteregcomp (set3 c) = C_bytereg c
byteregout (set3 c) s din = <<update_byte 3 s din, din>>
byteregstate (set3 c) s din = update_byte 3 s din

byteregdelay (set2 c) = #1
byteregcomp (set2 c) = C_bytereg c
byteregout (set2 c) s din = <<update_byte 2 s din, din>>
byteregstate (set2 c) s din = update_byte 2 s din

byteregdelay (set1 c) = #1
byteregcomp (set1 c) = C_bytereg c
byteregout (set1 c) s din = <<update_byte 1 s din, din>>
byteregstate (set1 c) s din = update_byte 1 s din

byteregdelay (set0 c) = #1
byteregcomp (set0 c) = C_bytereg c
byteregout (set0 c) s din = <<update_byte 0 s din, din>>
byteregstate (set0 c) s din = update_byte 0 s din
```

```
effect (A_bytereg a) = byteregeffect a
delay (A_bytereg a) = byteregdelay a
component (A_bytereg a) = byteregcomp a
byteregeffect a s c =
    S_bytereg <<byteregstate a (getbyteregstate (current s c)
                                                (bytereginput s c),
               byteregout a (getbyteregstate (current s c))
                                                  (bytereginput s c)>>
getbyteregstate (S_bytereg <<x,y>>) = x
getbyteregout0 (S_bytereg <<x,<<y0,y1>> >>)  = y0
getbyteregout1 (S_bytereg <<x,<<y0,y1>> >>)  = y1
is_proper_bytereg (S_bytereg <<x,<<y0,y1>> >>) = (! y0) & (! y1)
||clio: sy is_proper_bytereg extend
Goodbytereg 's' 'dataout' 'byteout':== 'dataout'='s' & 'good s'='True'

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class bitlatch.

bitlatch :: type
type bitlatch_localstate = <<BOOL,(BOOL)>>
LOCAL_STATE ::= S_bitlatch !bitlatch_localstate |+
ACTION ::= A_bitlatch !bitlatch_ACT |+
bitlatch_ACT ::=
 setb !bitlatch

bitlatchdelay (setb c) = #1
bitlatchcomp (setb c) = C_bitlatch c
bitlatchout (setb c) s in = in
bitlatchstate (setb c) s in = in

effect (A_bitlatch a) = bitlatcheffect a
delay (A_bitlatch a) = bitlatchdelay a
component (A_bitlatch a) = bitlatchcomp a
bitlatcheffect a s c =
   S_bitlatch <<bitlatchstate a (getbitlatchstate (current s c))
                                                  (bitlatchinput s c),
               bitlatchout a (getbitlatchstate (current s c))
                                                   (bitlatchinput s c)>>
getbitlatchstate (S_bitlatch <<x,y>>) = x
getbitlatchout0 (S_bitlatch <<x,y>>) = y
is_proper_bitlatch (S_bitlatch <<x,y>>) = ! y
||clio: sy is_proper_bitlatch extend
Goodbitlatch 's' 'out':== 's'='out'

ACTION ::= |
LOCAL_STATE ::= |
COMP ::=
 C_majority !majority |
 C_mux4 !mux4 |
 C_bytereg !bytereg |
 C_bitlatch !bitlatch

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Components (other than Controller and External)
|| and their connections.

majority ::= MAJ1 |+
majorityinput s (C_majority MAJ1) =
   <<getbyteregout0 (current s (C_bytereg R31)),
     getbyteregout0 (current s (C_bytereg R21)),
     getbyteregout0 (current s (C_bytereg PR1))>>
majority ::= MAJ2 |+
majorityinput s (C_majority MAJ2) =
```

```
    <<getbyteregout0 (current s (C_bytereg R32)),
      getbyteregout0 (current s (C_bytereg R12)),
      getbyteregout0 (current s (C_bytereg PR2))>>
majority ::= MAJ3 |+
majorityinput s (C_majority MAJ3) =
    <<getbyteregout0 (current s (C_bytereg R23)),
      getbyteregout0 (current s (C_bytereg R13)),
      getbyteregout0 (current s (C_bytereg PR3))>>
mux4 ::= MUX |+
mux4input s (C_mux4 MUX) =
    <<getbyteregout1 (current s (C_bytereg PR3)),
      getbyteregout1 (current s (C_bytereg PR2)),
      getbyteregout1 (current s (C_bytereg PR1)), bottom>>
bytereg ::= PR1 |+
bytereginput s (C_bytereg PR1) = external_input_prvt1 (current_input s)
bytereg ::= PR2 |+
bytereginput s (C_bytereg PR2) = external_input_prvt2 (current_input s)
bytereg ::= PR3 |+
bytereginput s (C_bytereg PR3) = external_input_prvt3 (current_input s)
bytereg ::= R12 |+
bytereginput s (C_bytereg R12) = external_input_vin1 (current_input s)
bytereg ::= R13 |+
bytereginput s (C_bytereg R13) = external_input_vin1 (current_input s)
bytereg ::= R21 |+
bytereginput s (C_bytereg R21) = external_input_vin2 (current_input s)
bytereg ::= R23 |+
bytereginput s (C_bytereg R23) = external_input_vin2 (current_input s)
bytereg ::= R31 |+
bytereginput s (C_bytereg R31) = external_input_vin3 (current_input s)
bytereg ::= R32 |+
bytereginput s (C_bytereg R32) = external_input_vin3 (current_input s)
bitlatch ::= VS |+
bitlatchinput s (C_bitlatch VS) = external_input_vstart (current_input s)
bitlatch ::= ST |+
bitlatchinput s (C_bitlatch ST) = getbitlatchout0 (current s (C_bitlatch VS))

majority ::= |
mux4 ::= |
bytereg ::= |
bitlatch ::= |
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Output function
sndng_controller_out s <<vstart, started>> = ready_to_send s
free_controller_out s <<vstart, started>> = voter_free s
Out s = (<<getmajorityout0 (current s (C_majority MAJ1)),
           getmajorityout0 (current s (C_majority MAJ2)),
           getmajorityout0 (current s (C_majority MAJ3)),
           <<getmajorityout1 (current s (C_majority MAJ1)),
             getmajorityout1 (current s (C_majority MAJ2)),
             getmajorityout1 (current s (C_majority MAJ3))>>,
           getmux4out0 (current s (C_mux4 MUX)),
           sndng_controller_out (getcontrolstate
                      (current s Controller)) (controllerinput s),
           free_controller_out (getcontrolstate (current s Controller))
                                                  (controllerinput s)>>)
external_output_m1 <<x0, x1, x2, x3, x4, x5, x6>> = x0
external_output_m2 <<x0, x1, x2, x3, x4, x5, x6>> = x1
external_output_m3 <<x0, x1, x2, x3, x4, x5, x6>> = x2
external_output_status <<x0, x1, x2, x3, x4, x5, x6>> = x3
external_output_cross <<x0, x1, x2, x3, x4, x5, x6>> = x4
```

```
external_output_sndng <<x0, x1, x2, x3, x4, x5, x6>> = x5
external_output_free <<x0, x1, x2, x3, x4, x5, x6>> = x6
inlist_from <<s,p,in>> = input_phases_of in

input_phases_of in = (map in input_phases) ++
                     input_phases_of (in. (#+ #num_phases))

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The scheduler.

scheduler LDP1 <<vstart, started>> 2 =
   ((vstart)->[A_bytereg(set1 PR1), A_bytereg(set1 PR2),
               A_bytereg(set1 PR3)];[A_bitlatch(setb VS)])
scheduler LDP2 <<vstart, started>> 2 =
    [A_bytereg(set3 PR1), A_bytereg(set3 PR2), A_bytereg(set3 PR3)]
scheduler XNG11 <<vstart, started>> 2 = (selprvt1byte1)
scheduler XNG12 <<vstart, started>> 2 = (selprvt1byte3)
scheduler XNG21 <<vstart, started>> 2 = (selprvt2byte1)
scheduler XNG22 <<vstart, started>> 2 = (selprvt2byte3)
scheduler XNG31 <<vstart, started>> 2 = (selprvt3byte1)
scheduler XNG32 <<vstart, started>> 2 = (selprvt3byte3)
scheduler OUT1 <<vstart, started>> 2 =
    [A_majority(select1 MAJ1), A_majority(select1 MAJ2),
     A_majority(select1 MAJ3)]
scheduler OUT2 <<vstart, started>> 2 =
    [A_majority(select3 MAJ1), A_majority(select3 MAJ2),
     A_majority(select3 MAJ3), A_bitlatch(setb VS)]
scheduler LDP1 <<vstart, started>> 0 =
    ((vstart)->[A_bytereg(set0 PR1), A_bytereg(set0 PR2),
     A_bytereg(set0 PR3)];[])++[A_bitlatch(setb ST)]
scheduler LDP2 <<vstart, started>> 0 =
    [A_bytereg(set2 PR1), A_bytereg(set2 PR2), A_bytereg(set2 PR3)]
scheduler XNG11 <<vstart, started>> 0 = (selprvt1byte0)
scheduler XNG12 <<vstart, started>> 0 = (selprvt1byte2)
scheduler XNG21 <<vstart, started>> 0 = (selprvt2byte0)
scheduler XNG22 <<vstart, started>> 0 = (selprvt2byte2)
scheduler XNG31 <<vstart, started>> 0 = (selprvt3byte0)
scheduler XNG32 <<vstart, started>> 0 = (selprvt3byte2)
scheduler OUT1 <<vstart, started>> 0 =
    [A_majority(select0 MAJ1), A_majority(select0 MAJ2),
     A_majority(select0 MAJ3)]
scheduler OUT2 <<vstart, started>> 0 =
    [A_majority(select2 MAJ1), A_majority(select2 MAJ2),
     A_majority(select2 MAJ3)]
scheduler XNG11 <<vstart, started>> 3 =
    [A_bytereg(set1 R12), A_bytereg(set1 R23)]++[Advance_controller]

scheduler XNG11 <<vstart, started>> 1 =
    [A_bytereg(set0 R12), A_bytereg(set0 R23)]
scheduler XNG12 <<vstart, started>> 1 =
    [A_bytereg(set2 R12), A_bytereg(set2 R23)]
scheduler XNG12 <<vstart, started>> 3 =
    [A_bytereg(set3 R12), A_bytereg(set3 R23)]++[Advance_controller]

scheduler XNG21 <<vstart, started>> 1 =
    [A_bytereg(set0 R13), A_bytereg(set0 R31)]
scheduler XNG21 <<vstart, started>> 3 =
    [A_bytereg(set1 R13), A_bytereg(set1 R31)]++[Advance_controller]

scheduler XNG22 <<vstart, started>> 1 =
    [A_bytereg(set2 R13), A_bytereg(set2 R31)]
scheduler XNG22 <<vstart, started>> 3 =
    [A_bytereg(set3 R13), A_bytereg(set3 R31)]++[Advance_controller]
```

```
scheduler XNG31 <<vstart, started>> 1 =
    [A_bytereg(set0 R21), A_bytereg(set0 R32)]
scheduler XNG31 <<vstart, started>> 3 =
    [A_bytereg(set1 R21), A_bytereg(set1 R32)]++[Advance_controller]

scheduler XNG32 <<vstart, started>> 1 =
    [A_bytereg(set2 R21), A_bytereg(set2 R32)]
scheduler XNG32 <<vstart, started>> 3 =
    [A_bytereg(set3 R21), A_bytereg(set3 R32)]++[Advance_controller]

selprvt1byte0 = [A_bytereg(read0 PR1), A_mux4(choose3 MUX)]
selprvt1byte1 = [A_bytereg(read1 PR1), A_mux4(choose3 MUX)]
selprvt1byte2 = [A_bytereg(read2 PR1), A_mux4(choose3 MUX)]
selprvt1byte3 = [A_bytereg(read3 PR1), A_mux4(choose3 MUX)]
selprvt2byte0 = [A_bytereg(read0 PR2), A_mux4(choose2 MUX)]
selprvt2byte1 = [A_bytereg(read1 PR2), A_mux4(choose2 MUX)]
selprvt2byte2 = [A_bytereg(read2 PR2), A_mux4(choose2 MUX)]
selprvt2byte3 = [A_bytereg(read3 PR2), A_mux4(choose2 MUX)]
selprvt3byte0 = [A_bytereg(read0 PR3), A_mux4(choose1 MUX)]
selprvt3byte1 = [A_bytereg(read1 PR3), A_mux4(choose1 MUX)]
selprvt3byte2 = [A_bytereg(read2 PR3), A_mux4(choose1 MUX)]
selprvt3byte3 = [A_bytereg(read3 PR3), A_mux4(choose1 MUX)]
scheduler CMPP <<vstart, started>> 0 =
   [A_majority(comp MAJ1), A_majority(comp MAJ2), A_majority(comp MAJ3)]
scheduler s in 3 = [Advance_controller]
scheduler s in t = []

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Proper state predicates.

is_proper_state `s` :==
`is_proper_contr (current s Controller)` = `True`
& `is_proper_majority (current s (C_majority MAJ1))` = `True`
& `is_proper_majority (current s (C_majority MAJ2))` = `True`
& `is_proper_majority (current s (C_majority MAJ3))` = `True`
& `is_proper_mux4 (current s (C_mux4 MUX))` = `True`
& `is_proper_bytereg (current s (C_bytereg PR1))` = `True`
& `is_proper_bytereg (current s (C_bytereg PR2))` = `True`
& `is_proper_bytereg (current s (C_bytereg PR3))` = `True`
& `is_proper_bytereg (current s (C_bytereg R12))` = `True`
& `is_proper_bytereg (current s (C_bytereg R13))` = `True`
& `is_proper_bytereg (current s (C_bytereg R21))` = `True`
& `is_proper_bytereg (current s (C_bytereg R23))` = `True`
& `is_proper_bytereg (current s (C_bytereg R31))` = `True`
& `is_proper_bytereg (current s (C_bytereg R32))` = `True`
& `is_proper_bitlatch (current s (C_bitlatch VS))` = `True`
& `is_proper_bitlatch (current s (C_bitlatch ST))` = `True`

Proper_state `s` :=
`pending_changes s`=`[]`
& is_proper_state `s`
& Goodmajority `getmajoritystate(current s (C_majority MAJ1))`
                    `getmajorityout0(current s (C_majority MAJ1))`
                    `getmajorityout1(current s (C_majority MAJ1))`
& Goodmajority `getmajoritystate(current s (C_majority MAJ2))`
                    `getmajorityout0(current s (C_majority MAJ2))`
                    `getmajorityout1(current s (C_majority MAJ2))`
& Goodmajority `getmajoritystate(current s (C_majority MAJ3))`
                    `getmajorityout0(current s (C_majority MAJ3))`
                    `getmajorityout1(current s (C_majority MAJ3))`
```

```
 & Goodbytereg 'getbyteregstate(current s (C_bytereg PR1))'
                        'getbyteregout0(current s (C_bytereg PR1))'
                        'getbyteregout1(current s (C_bytereg PR1))'
 & Goodbytereg 'getbyteregstate(current s (C_bytereg PR2))'
                        'getbyteregout0(current s (C_bytereg PR2))'
                        'getbyteregout1(current s (C_bytereg PR2))'
 & Goodbytereg 'getbyteregstate(current s (C_bytereg PR3))'
                        'getbyteregout0(current s (C_bytereg PR3))'
                        'getbyteregout1(current s (C_bytereg PR3))'
 & Goodbytereg 'getbyteregstate(current s (C_bytereg R12))'
                        'getbyteregout0(current s (C_bytereg R12))'
                        'getbyteregout1(current s (C_bytereg R12))'
 & Goodbytereg 'getbyteregstate(current s (C_bytereg R13))'
                        'getbyteregout0(current s (C_bytereg R13))'
                        'getbyteregout1(current s (C_bytereg R13))'
 & Goodbytereg 'getbyteregstate(current s (C_bytereg R21))'
                        'getbyteregout0(current s (C_bytereg R21))'
                        'getbyteregout1(current s (C_bytereg R21))'
 & Goodbytereg 'getbyteregstate(current s (C_bytereg R23))'
                        'getbyteregout0(current s (C_bytereg R23))'
                        'getbyteregout1(current s (C_bytereg R23))'
 & Goodbytereg 'getbyteregstate(current s (C_bytereg R31))'
                        'getbyteregout0(current s (C_bytereg R31))'
                        'getbyteregout1(current s (C_bytereg R31))'
 & Goodbytereg 'getbyteregstate(current s (C_bytereg R32))'
                        'getbyteregout0(current s (C_bytereg R32))'
                        'getbyteregout1(current s (C_bytereg R32))'
 & Goodbitlatch 'getbitlatchstate(current s (C_bitlatch VS))'
                        'getbitlatchout0(current s (C_bitlatch VS))'
 & Goodbitlatch 'getbitlatchstate(current s (C_bitlatch ST))'
                        'getbitlatchout0(current s (C_bitlatch ST))'
 & Proper_External 's'

||clio: symbol Proper_state extend_auto
||clio: symbol pending_changes extend_auto
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| For export to other modules, we define functions
|| access the internal state of each component that has one.
get_MAJ1_state s = getmajoritystate (current s (C_majority MAJ1))
get_MAJ2_state s = getmajoritystate (current s (C_majority MAJ2))
get_MAJ3_state s = getmajoritystate (current s (C_majority MAJ3))
get_PR1_state s = getbyteregstate (current s (C_bytereg PR1))
get_PR2_state s = getbyteregstate (current s (C_bytereg PR2))
get_PR3_state s = getbyteregstate (current s (C_bytereg PR3))
get_R12_state s = getbyteregstate (current s (C_bytereg R12))
get_R13_state s = getbyteregstate (current s (C_bytereg R13))
get_R21_state s = getbyteregstate (current s (C_bytereg R21))
get_R23_state s = getbyteregstate (current s (C_bytereg R23))
get_R31_state s = getbyteregstate (current s (C_bytereg R31))
get_R32_state s = getbyteregstate (current s (C_bytereg R32))
get_VS_state s = getbitlatchstate (current s (C_bitlatch VS))
get_ST_state s = getbitlatchstate (current s (C_bitlatch ST))

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Auxillary definitions.

type bitvec = [BOOL]
```

```
dataof <<data, bit>> = data
bitof <<data, bit>> = bit

|| ** Should this be majority_does_not_exist ? **
majority_exists in1 in2 in3 = (in1 = in2) | (in1 = in3) | (in2 = in3)
majority_of in1 in2 in3 = (in1=in2 -> in1; (in1=in3 -> in1; in2))

isbitvector n bitvec = (n = ## bitvec)

ready_to_send s = (s=OUT1)|(s=OUT2)
voter_free s  = s=LDP1

from_proc <<p1,p2,p3,v1,v2,v3,go>> = <<p1,p2,p3,go>>
from_vtrs <<p1,p2,p3,v1,v2,v3,go>> = <<v3,v2,v1>>

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The cutpoints and loop conditions.

CUTPOINT ::= start

at_start s = start_state (controllerstate s)

start_state LDP1 = True
start_state x = False

PATH ::= startPATH0 |+
path_start startPATH0 = start
path_end startPATH0 = start
path_length startPATH0 = #11

path_condition `startPATH0` `s` :==
TRUE
& `controllerstate s`=`LDP1`
& `~(startedof (controllerinput (iterate Execute (#1) s)))`=`False`
& `controllerstate (iterate Execute (#1) s)`=`LDP2`
& `controllerstate (iterate Execute (#2) s)`=`XNG11`
& `controllerstate (iterate Execute (#3) s)`=`XNG12`
& `controllerstate (iterate Execute (#4) s)`=`XNG21`
& `controllerstate (iterate Execute (#5) s)`=`XNG22`
& `controllerstate (iterate Execute (#6) s)`=`XNG31`
& `controllerstate (iterate Execute (#7) s)`=`XNG32`
& `controllerstate (iterate Execute (#8) s)`=`CMPP`
& `controllerstate (iterate Execute (#9) s)`=`OUT1`
& `controllerstate (iterate Execute (#10) s)`=`OUT2`

PATH ::= startPATH1 |+
path_start startPATH1 = start
path_end startPATH1 = start
path_length startPATH1 = #1

path_condition `startPATH1` `s` :==
TRUE
& `controllerstate s`=`LDP1`
& `~(startedof (controllerinput (iterate Execute (#1) s)))`=`True`


Invariant `p` `s` := TRUE
PATH ::= |


Advance_Relation `p1` `p2` `s1` `s2` := TRUE
path_precond `path` `s` :=
  `!path`=`True`
  & path_condition `path` `s`
  & Invariant `path_start path` `s`
||clio: sy path_precond extend
||clio: sy Invariant extend

path_postcond `path` `s` :=
```

```
   Invariant 'path_end path' 'iterate Execute (path_length path) s'
   & Advance_Relation 'path_start path' 'path_end path'
                      's' 'iterate Execute (path_length path) s'

VC 'path' 's':=
 path_precond 'path' 's' => path_postcond 'path' 's'

VC_ok 's' :==  (path) VC 'path' 's'
VC_ok_lemma :==  Proper_state 's' => VC_ok 's'
Timing_ok 's' :==
   Proper_state 'Execute s'
Timing_ok_lemma :==
   Timing_ok 's', Proper_state 's'

Timing_ok_case 's':==
   Timing_ok 's', 'controllerstate s'='c'

Timing_ok_by_cases :== Proper_state 's' => Timing_ok_case 's'
```

4 FtCayuga Specification

In the following the functions that are used in both the abstract as well as the design specifications are given in a separate section.

4.1 Abstract Specification

```
FROM CommonTheorySec IMPORT data_to_addr, data_to_regaddr, byte_num,byte_inc,
                    good, reset_to_addr, inc_data, LSB,
                    JIF,JIT,JMP, SADD,MOVE,ICOP,PVT,STATUS,VT1,VT2,
                    VT3, BYTE0, dstof,opclassof, opcodeof,  update,protected,
                    data, regaddr, addr, sregaddr, data_to_sregaddr DATUM,
                    Word, set_byte, data_false, getbyte, get_byte, byte
FROM CommonPartSec IMPORT   current_result, interruptof,
FROM FtCayugaDesignSec IMPORT  Execute, Proper_state, get_HAND_state,
                           controllerstate, get_NXPC_state, get_RSLT_state,
                           get_DST_state, get_IREG_state, get_MEM_state,
                           get_REG_state, get_INST_state, get_SREG_state,
                           get_BC_state,inlist_from,
                           WBA,WBS,WBJ,WBL,DF,DFI,INTF,INTC,OP_switch,
                           STATE,CONTROLSTATE,EXTSTATE, time_abs, vtrs_update

|| Now we define a partial abstraction, FtCayugaABS, of the ftCayuga state
|| and a corresponding version, FtCayugaStep, of the Execute function.

type FTCAYUGASTATE = <<CONTROLSTATE,data,data,data,data,
                       addr ->data,data,regaddr ->data,data,
                       sregaddr->data,byte_num>>

FtCayugaABS s =
 <<controllerstate s, get_IREG_state s, get_DST_state s,
   get_RSLT_state s, get_HAND_state s, get_MEM_state s,
   get_NXPC_state s, get_REG_state s, get_INST_state s,
   get_SREG_state s, get_BC_state s>>

FtInputABS s = time_abs(inlist_from s)
```

```
|| Selector functions on the FTCAYUGASTATE:
controf <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = cs
iregof <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = irg
dst_of <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = dst
rsltof <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = rslt
handof <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = hnd
memof <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = mem
nxpcof <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = nx
regof <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = reg
inst_of <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = inst
sregof <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = sreg
bytecount <<cs,irg,dst,rslt,hnd,mem,nx,reg,inst,sreg,bc>> = bc

FtCayugaStep s in =
    <<newcontr (controf s) (interruptof in) (iregof s),
      newireg (controf s) (interruptof in) s,
      newdst (controf s) (dst_of s)(iregof s),
      newrslt (controf s) s,
      newhand (controf s) (interruptof in) (handof s),
      newmem (controf s) s,
      newnxpc (controf s) (interruptof in) s,
      newreg (controf s) s,
      newinst (controf s) (inst_of s)(iregof s),
      newsreg (controf s) in s,
      newbc s>>

||******************** next control state *********************************
newcontr WBS in irg = in->DFI;DF
newcontr WBL in irg = in->DFI;DF
newcontr DFI in irg =  INTF
newcontr INTF in irg =  INTC
newcontr INTC in irg = active_nextstate False irg
newcontr cs in irg = active_nextstate in irg

active_nextstate in ireg =
     in->INTF;(OP_switch (opclassof (opcodeof ireg)))

||******************** next ireg state *********************************
newireg WBS in s = iregof s
newireg WBL in s = iregof s
newireg WBA in s = memof s (data_to_addr(next_nxpc in (nxpcof s)))
newireg WBJ in s = memof s (data_to_addr (jmpnxpc in s))
newireg DFI in s = memof s (data_to_addr (handof s))
newireg DF in s = memof s (data_to_addr(next_nxpc in (nxpcof s)))
newireg INTF in s = memof s (data_to_addr (iregof s))
newireg INTC in s = memof s (data_to_addr(inc_data (nxpcof s)))

jmpnxpc in s =
   in->(reset_to_addr in)
   (jump_cond (inst_of s)(regof s)(dst_of s))-> (rsltof s); (inc_data (nxpcof s))

jump_cond inst reg dst =
       ((opcodeof inst)=JMP)
               xor (((opcodeof inst)=JIT) & (LSB(reg (data_to_regaddr dst))))
               xor (((opcodeof inst)=JIF) & ~(LSB(reg (data_to_regaddr dst))))

||******************** next dst state *********************************
newdst WBS dst ireg = dst
newdst WBL dst ireg = dst
newdst INTF dst ireg = dst
newdst cs dst ireg = dstof (ireg)

||******************** next rslt state *********************************
newrslt WBS s = rsltof s
newrslt WBL s = rsltof s
```

```
newrslt INTF s = rsltof s
newrslt WBA s =
    current_result <<memof s,nxpcof s,wba_reg s,iregof s,sregof s,[]>>
newrslt cs s =
    current_result <<memof s,nxpcof s,regof s,iregof s,sregof s,[]>>

wba_reg s = regof s , opcodeof (inst_of s) = SADD
          update (regof s) (data_to_regaddr (dst_of s)) (rsltof s)

||******************** next hand state *********************************
newhand DFI in h = h
newhand INTF in h = h
newhand cs in h = reset_to_addr in

||******************** next mem state **********************************
newmem WBS s = update (memof s)(data_to_addr (rsltof s))
                      (regof s (data_to_regaddr (dst_of s)))
newmem cs s = memof s

||******************** next nxpc state *********************************
newnxpc WBS in s = nxpcof s
newnxpc WBL in s = nxpcof s
newnxpc WBA in s = next_nxpc in (nxpcof s)
newnxpc WBJ in s = jmpnxpc in s
newnxpc DFI in s = handof s
newnxpc DF in s =  next_nxpc in (nxpcof s)
newnxpc INTF in s = iregof s
newnxpc INTC in s = inc_data (nxpcof s)

next_nxpc in nxpc = in-> (reset_to_addr in); (inc_data nxpc)

||******************** next reg state **********************************
newreg WBL s =
     update (regof s) (data_to_regaddr (dst_of s))
            (memof s (data_to_addr (rsltof s)))
newreg WBA s = wba_reg s
newreg cs s = regof s

||******************** next inst state *********************************
newinst INTF inst ireg = inst
newinst cs inst ireg = ireg

||******************** next sreg state *********************************
newsreg WBA in s =
     update (special_update s in) (data_to_sregaddr (dst_of s)) (rsltof s)
       , (opcodeof (inst_of s) = SADD)& ~(protected(data_to_sregaddr (dst_of s)))
     special_update s in
newsreg cs in s = special_update s in

special_update s <<r,v1,v2,v3,[st1,st2]>> =
  ic_update (sregof s) (bytecount s) v1 v2 v3 , ic_data_ready st1
  status_update (sregof s) st1 , ic_cycle_completed (sregof s) st1
  status_clear (sregof s), (voter_free st1) & (opcodeof (inst_of s) = ICOP)
  sregof s
special_update s in = sregof s

msbof, next2msbof :: byte -> BOOL
ic_data_ready st1 = ~(msbof st1) & (next2msbof st1)
ic_cycle_completed sreg st1 = (msbof st1) & ~(msbof (get_byte 3 (sreg STATUS)))
voter_free st1  = (msbof st1)

|| status_clear resets the msb of (sreg STATUS)
status_clear sreg = update sreg STATUS data_false

|| status update sreg updates the STATUS register in sreg to st1
status_update sreg st1 = update sreg STATUS (DATUM (Word st1 st1 st1 st1))
```

```
|| ic_update updates the three ic registers in sreg
ic_update sreg b v1 v2 v3 =
    update (update (update sreg VT1 (update_bytes b (sreg VT1) v1))
                                VT2 (update_bytes b (sreg VT2) v2))
                                VT3 (update_bytes b (sreg VT3) v3)
byte_inc2 = byte_inc.byte_inc
byte_inc3 = byte_inc.byte_inc2
update_bytes b w [v1,v2] =
     set_byte b (set_byte (byte_inc3 b) w v1) v2

||******************** next bc state *********************************
newbc s = byte_inc(byte_inc (bytecount s))
newbc s = byte_inc(byte_inc (bytecount s))


||*************************************************************************
|| the output functions
||clio: add vstartout

vstartout s =
    fourphase (ic_initiate (controf s) (inst_of s) (sregof s STATUS))

stat_byte <<r,v1,v2,v3,st>> = hd st

ic_initiate cstate inst st =
        (cstate = WBA) & (opcodeof inst = ICOP) & (msbof (getbyte BYTE0 st))


prvtout s = pad [(getbyte  (byte_inc (bytecount s)) (sregof s PVT)),
                 (getbyte  (byte_inc (byte_inc (bytecount s))) (sregof s PVT))]

pad [a,b] = <<a,a,b,b>>
fourphase a = <<a,a,a,a>>


||*************************************************************************
|| Properness predicate:

Goodmem `s` := (x::addr)`good (s x)`=`!!x` & `!(s x)`=`!!x`
Goodregfile `s` := (x::regaddr)`good(s x)`=`!x` & `!(s x)`=`!x`
Goodsregfile `s` := (x::sregaddr)`good(s x)`=`!x` & `!(s x)`=`!x`

Proper_ftcayuga `<<cstate,ireg,dst,rslt,hand,mem,nxpc,reg,inst,sreg,bc>>` :=
`!cstate` = `True`
& `good ireg` = `True`
& `good dst` = `True`
& `good rslt` = `True`
& `good hand` = `True`
& Goodmem `mem`
& `good nxpc` = `True`
& Goodregfile `reg`
& `good inst` = `True`
& Goodsregfile `sreg`
& `!bc` = `True`

good_ftcayuga_in <<r,v1,v2,v3,[st1,st2]>> =
    !r & ((ic_data_ready st1)->(good_voter_vals v1 v2 v3);True)

good_voter_vals [v11,v12] [v21,v22] [v31,v32] =
     !v11 & !v12 & !v21 & !v22 & !v31 & !v32

Proper_ftcayuga_lemma :=
    (Proper_ftcayuga `s `  & `good_ftcayuga_in in`=`True`)
      => Proper_ftcayuga `FtCayugaStep s in`

|| Stuff we need about the function "statusword"
type threebit = <<BOOL,BOOL,BOOL>>
stwd1,stwd2 :: <<threebit,threebit>>->><<BOOL,BOOL>>->><<BOOL,BOOL>>->byte
```

```
statusword x y z = [stwd1 x y z, stwd2 x y z]


|| Some lemmas about the PVT register
newPVT s =
 ((opcodeof (inst_of s) = SADD)& ((data_to_sregaddr (dst_of s))=PVT)
    & ((controf s)=WBA))
   -> (rsltof s); sregof s PVT


PVT_lemma1 :== 'ic_update sreg b v1 v2 v3 PVT'='sreg PVT'
PVT_lemma2 :==
 'special_update s <<r,v1,v2,v3,[stwd1 x <<fr,y>> <<snd,z>>,st2]>> PVT'
    ='sregof s PVT'
              , '!fr & !snd & !(inst_of s) & (good (sregof s STATUS))'='True'

PVT_lemma3 :==
 'sregof (FtCayugaStep s <<r,v1,v2,v3,[stwd1 x <<fr,y>> <<snd,z>>,st2]>>) PVT'
   = 'newPVT s'
    , '!fr & !snd & !(inst_of s)& !(dst_of s) & (good (sregof s STATUS))'='True'

newPVT2 s in =
         ((opcodeof (newinst (controf s) (inst_of s)(iregof s)) = SADD)
           & ((data_to_sregaddr (newdst (controf s) (dst_of s)(iregof s)))=PVT)
            & ((newcontr (controf s) in (iregof s))=WBA))
           -> (newrslt (controf s) s); newPVT s
PVT_lemma4 :=
 'newPVT (FtCayugaStep s <<r,v1,v2,v3,[stwd1 x <<fr,y>> <<snd,z>>,st2]>>)'
     = 'newPVT2 s r'
       , '!fr & !snd & !(inst_of s)& !(dst_of s) & (good (sregof s STATUS))'='True'

|| Some lemmas about the VT1, VT2, VT3 registers

VT123_lemma1 :==
  'newsreg cs in s v'='special_update s in v'
    , ('v'='VT1'\/ 'v'='VT2' \/ 'v'='VT3')
          & '!cs & !(inst_of s) & !(dst_of s)'='True'
```

4.2 Design Specification

```
FROM CommonTheorySec IMPORT data,byte,nodata,opcode,addr,regaddr,sregaddr,
                        good, opcodeof, indirect, src1of,src2of,dstof,
                        data_to_addr, data_to_regaddr, data_to_sregaddr,
                        reset_to_addr,  regaddr_to_sregaddr, byte_num,
                        update, inc_data, add_data, sub_data, cne_data, LSB,
                        ACLASS,JCLASS,SCLASS,LCLASS,opclassof
                        LD, ST, ADD, JMP, JIT, JIF, CNE, MOVE, SADD,
                        PVT,VT1,VT2,VT3,STATUS,protected,
                        BYTE0, set_byte, getbyte, byte_inc
FROM CommonPartSec IMPORT   Step, prefetch, current_opclass,
                        current_dst, current_result
||clio: add OP_switch
||clio: modify "OP_switch" off
||clio: add vtrs_update
||clio: modify "vtrs_update" off

|| Generated by the spectool
||clio: symbol Execute never
||clio: mod * off
||clio: add *
||clio: mod * on
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
```

```
|| The Controller.

COMP ::= Controller |+
ACTION ::= Advance_controller|+
CONTROLSTATE :: type
LOCAL_STATE ::= S_Controller !CONTROLSTATE |+
getcontrolstate (S_Controller x) = x
is_proper_contr (S_Controller x) = !x
||clio: sy is_proper_contr extend
controllerstate s = getcontrolstate (current s Controller)
effect Advance_controller s c =
        S_Controller (nextstate (controllerstate s) (controllerinput s))
delay Advance_controller = #0
component Advance_controller = Controller

controllerinput s =
   <<getbitlatchout0 (current s (C_bitlatch RESET)),
     opcodeof(getlatchout0 (current s (C_latch INST))),
     getdecoderout0 (current s (C_decoder DEC)),
     getdecoderout1 (current s (C_decoder DEC)),
     getmatcherout0 (current s (C_matcher MTCH)),
     getmatcherout1 (current s (C_matcher MTCH)),
     getregfileout2 (current s (C_regfile REG))>>
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The Control states and the next_state function.

resetof <<reset, inst, op, ind, eq1, eq2, cc>> = reset
instof <<reset, inst, op, ind, eq1, eq2, cc>> = inst
opof <<reset, inst, op, ind, eq1, eq2, cc>> = op
indof <<reset, inst, op, ind, eq1, eq2, cc>> = ind
eq1of <<reset, inst, op, ind, eq1, eq2, cc>> = eq1
eq2of <<reset, inst, op, ind, eq1, eq2, cc>> = eq2
ccof <<reset, inst, op, ind, eq1, eq2, cc>> = cc

CONTROLSTATE ::= WBS |+
nextstate WBS in = ((resetof in))->(DFI);(DF)
CONTROLSTATE ::= WBL |+
nextstate WBL in = ((resetof in))->(DFI);(DF)
CONTROLSTATE ::= WBA |+
nextstate WBA in = ((resetof in))->(INTF);(OP_switch (opclassof (opof in)))
CONTROLSTATE ::= WBJ |+
nextstate WBJ in = ((resetof in))->(INTF);(OP_switch (opclassof (opof in)))
OP_switch x =
WBA, x = ACLASS
WBS, x = SCLASS
WBL, x = LCLASS
WBJ
CONTROLSTATE ::= DFI |+
nextstate DFI in = INTF
CONTROLSTATE ::= DF |+
nextstate DF in = ((resetof in))->(INTF);(OP_switch (opclassof (opof in)))
CONTROLSTATE ::= INTF |+
nextstate INTF in = INTC
CONTROLSTATE ::= INTC |+
nextstate INTC in = OP_switch (opclassof (opof in))
CONTROLSTATE ::= |
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The component External.

type EXTSTATE = <<BOOL, byte, byte, byte, byte>>

external_input_reset <<x0, x1, x2, x3, x4>> = x0
external_input_v1 <<x0, x1, x2, x3, x4>> = x1
external_input_v2 <<x0, x1, x2, x3, x4>> = x2
external_input_v3 <<x0, x1, x2, x3, x4>> = x3
```

```
external_input_ST <<x0, x1, x2, x3, x4>> = x4
is_proper_ext <<x0, x1, x2, x3, x4>> = (!x0) & (!x1) & (!x2) & (!x3) & (!x4)
||clio: sy is_proper_ext extend

current_input <<s,p,in>> = in Zero
nth_input n <<s,p,in>> = in n
||clio: sy Proper_External extend
Proper_External `<<s,p,in>>` :=
(t::NAT) `is_proper_ext (in t)`=`True`
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| There are 4 clock phases per cycle.

num_phases = 4
input_phases  = [#0, #1, #3]

output_phase 1 = True
output_phase 3 = True
output_phase n = False

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The generic Execute function.

type SYSTEM_STATE = COMP->LOCAL_STATE
type INPUT_STREAM = NAT->EXTSTATE
type CHANGE = <<COMP,NAT,LOCAL_STATE>>
type STATE = <<SYSTEM_STATE, [CHANGE], INPUT_STREAM>>

pending_changes <<s,p,in>> = p

current <<s,p,in>> c = bottom, pending p c
                       s c

pending [] c = False
pending (<<c,t,v>>:rest) c = True
pending (<<c2,t,v>>:rest) c = pending rest c

Execute s = do_phases 0 s


do_phases n s = update_state s , n = num_phases
                do_phases (n+1) (do_phase n s)

Output s = generate_output 0 s


generate_output n s {n=num_phases} = []
generate_output n s {output_phase n} =
                  Out(do_phase n s) : generate_output (n+1) (do_phase n s)
generate_output n s = generate_output (n+1) (do_phase n s)

||clio: modify_rule "do_phases" count 20000
||clio: symbol do_phases never

update_state <<s,p,in>> = do_changes p <<s,[],in>>

do_phase n <<s,p,in>> =
  advance_inputstream (do_actions (current_schedule s2 n) s2)
  where s2 = update_state <<s,p,in>>

current_schedule s n = scheduler (controllerstate s) (controllerinput s) n

do_change <<c,Zero,v>> <<s,p,in>> = <<update s c v, p, in>>
do_change <<c,Succ n,v>> <<s,p,in>> = <<s, <<c,n,v>>:p, in>>

do_action a s = do_change (change_of a s) s
```

```
advance_inputstream <<s,p,in>> = <<s,p,in.Succ>>

change_of a s = <<component a, delay a, effect a s (component a)>>

do_changes [] s = s
do_changes (<<c,t,v>>:rest) s = do_changes rest (do_change <<c,t,v>> s)
|| This is a trick to handle conditional actions correctly
do_actions :: [ACTION]->STATE->STATE
AXIOM (s)'do_actions [] s' = 's'
AXIOM (a)(rest)(s)'do_actions (a:rest) s' = 'do_actions rest (do_action a s)'
update s c v c2 = (c=c2)->v; s c2

foldl op s [] = s
foldl op s (a:rest) = foldl op (op a s) rest

map f [] = []
map f (a:x) = (f a): (map f x)
list_to Zero = []
list_to (Succ n) = list_to n ++ [n]
iterate f Zero s = s
iterate f (Succ n) s = iterate f n (f s)
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Component Classes.

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class latch.

latch :: type
type latch_localstate = <<data,data>>
LOCAL_STATE ::= S_latch !latch_localstate |+
ACTION ::= A_latch !latch_ACT |+
latch_ACT ::=
 set !latch

latchdelay (set c) = #1
latchcomp (set c) = C_latch c
latchout (set c) s in = in
latchstate (set c) s in = in

effect (A_latch a) = latcheffect a
delay (A_latch a) = latchdelay a
component (A_latch a) = latchcomp a
latcheffect a s c =
   S_latch <<latchstate a (getlatchstate (current s c))
                                                (latchinput s c),
             latchout a (getlatchstate (current s c)) (latchinput s c)>>
getlatchstate (S_latch <<x,y>>) = x
getlatchout0 (S_latch <<x,y>>) = y
is_proper_latch (S_latch <<x,y>>) = good y
||clio: sy is_proper_latch extend
Goodlatch 's' 'out':== 's'='out'

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class mux.

mux :: type
type mux_localstate = data
LOCAL_STATE ::= S_mux !mux_localstate |+
ACTION ::= A_mux !mux_ACT |+
mux_ACT ::=
 choose3 !mux |
 choose2 !mux |
 choose1 !mux

muxdelay (choose3 c) = #0
muxcomp (choose3 c) = C_mux c
muxout (choose3 c) <<in1,in2,in3>> = in3
```

```
muxdelay (choose2 c) = #0
muxcomp (choose2 c) = C_mux c
muxout (choose2 c) <<in1,in2,in3>> = in2

muxdelay (choose1 c) = #0
muxcomp (choose1 c) = C_mux c
muxout (choose1 c) <<in1,in2,in3>> = in1

effect (A_mux a) = muxeffect a
delay (A_mux a) = muxdelay a
component (A_mux a) = muxcomp a
muxeffect a s c = S_mux (muxout a (muxinput s c))
getmuxout0 (S_mux y) = y
is_proper_mux (S_mux y) = good y
||clio: sy is_proper_mux extend

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class mem.

mem :: type
type mem_localstate = <<addr->data,data>>
LOCAL_STATE ::= S_mem !mem_localstate |+
ACTION ::= A_mem !mem_ACT |+
mem_ACT ::=
 write !mem |
 read !mem

memdelay (write c) = #2
memcomp (write c) = C_mem c
memout (write c) s <<aval,dval>> = nodata
memstate (write c) s <<aval,dval>> = update s aval dval

memdelay (read c) = #2
memcomp (read c) = C_mem c
memout (read c) s <<aval,dval>> = s aval
memstate (read c) s <<aval,dval>> = s

effect (A_mem a) = memeffect a
delay (A_mem a) = memdelay a
component (A_mem a) = memcomp a
memeffect a s c =
   S_mem <<memstate a (getmemstate (current s c)) (meminput s c),
           memout a (getmemstate (current s c)) (meminput s c)>>
getmemstate (S_mem <<x,y>>) = x
getmemout0 (S_mem <<x,y>>) = y
is_proper_mem (S_mem <<x,y>>) = ! y
||clio: sy is_proper_mem extend
Goodmem 's' 'out':== (x::addr)'good (s x)'='!x' & '!(s x)'='!x'

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class regfile.

regfile :: type
type regfile_localstate = <<regaddr->data,<<data,data,BOOL>>>>
LOCAL_STATE ::= S_regfile !regfile_localstate |+
ACTION ::= A_regfile !regfile_ACT |+
regfile_ACT ::=
 load !regfile |
 d_unload !regfile |
 unload !regfile

regfiledelay (load c) = #1
regfilecomp (load c) = C_regfile c
```

```
regfileout (load c) s <<src1,src2,dst,dval>> = <<nodata, nodata, False>>
regfilestate (load c) s <<src1,src2,dst,dval>> = update s dst dval

regfiledelay (d_unload c) = #1
regfilecomp (d_unload c) = C_regfile c
regfileout (d_unload c) s <<src1,src2,dst,dval>> = <<s dst, nodata, False>>
regfilestate (d_unload c) s <<src1,src2,dst,dval>> = s

regfiledelay (unload c) = #1
regfilecomp (unload c) = C_regfile c
regfileout (unload c) s <<src1,src2,dst,dval>> =
                          <<s src1, s src2, LSB (s dst)>>
regfilestate (unload c) s <<src1,src2,dst,dval>> = s

effect (A_regfile a) = regfileeffect a
delay (A_regfile a) = regfiledelay a
component (A_regfile a) = regfilecomp a
regfileeffect a s c =
   S_regfile <<regfilestate a (getregfilestate (current s c))
                                              (regfileinput s c),
              regfileout a (getregfilestate (current s c))
                                                (regfileinput s c)>>

getregfilestate (S_regfile <<x,y>>) = x
getregfileout0 (S_regfile <<x,<<y0,y1,y2>> >>)  = y0
getregfileout1 (S_regfile <<x,<<y0,y1,y2>> >>)  = y1
getregfileout2 (S_regfile <<x,<<y0,y1,y2>> >>)  = y2
is_proper_regfile (S_regfile <<x,<<y0,y1,y2>> >>) = (! y0) & (! y1) & (! y2)
||clio: sy is_proper_regfile extend
Goodregfile `s` `op1` `op2` `cc`:==
              (x::regaddr)`good(s x)`=`!x` & `!(s x)`=`!x`

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class decoder.

decoder :: type
type decoder_localstate = <<opcode,BOOL,data,regaddr,data>>
LOCAL_STATE ::= S_decoder !decoder_localstate |+
ACTION ::= A_decoder !decoder_ACT |+
decoder_ACT ::=
 decode !decoder

decoderdelay (decode c) = #0
decodercomp (decode c) = C_decoder c
decoderout (decode c) in =
        <<opcodeof in, indirect in, dstof in, src1of in, src2of in>>

effect (A_decoder a) = decodereffect a
delay (A_decoder a) = decoderdelay a
component (A_decoder a) = decodercomp a
decodereffect a s c = S_decoder (decoderout a (decoderinput s c))
getdecoderout0 (S_decoder <<y0,y1,y2,y3,y4>>)  = y0
getdecoderout1 (S_decoder <<y0,y1,y2,y3,y4>>)  = y1
getdecoderout2 (S_decoder <<y0,y1,y2,y3,y4>>)  = y2
getdecoderout3 (S_decoder <<y0,y1,y2,y3,y4>>)  = y3
getdecoderout4 (S_decoder <<y0,y1,y2,y3,y4>>)  = y4
is_proper_decoder (S_decoder <<y0,y1,y2,y3,y4>>) =
                  (! y0) & (! y1) & (good y2) & (! y3) & (good y4)
||clio: sy is_proper_decoder extend


||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class alu.

alu :: type
```

```
type alu_localstate = data
LOCAL_STATE ::= S_alu !alu_localstate |+
ACTION ::= A_alu !alu_ACT |+
alu_ACT ::=
 compare !alu |
 subtract !alu |
 add !alu

aludelay (compare c) = #2
alucomp (compare c) = C_alu c
aluout (compare c) <<op1,op2>> = cne_data op1 op2

aludelay (subtract c) = #2
alucomp (subtract c) = C_alu c
aluout (subtract c) <<op1,op2>> = sub_data op1 op2

aludelay (add c) = #2
alucomp (add c) = C_alu c
aluout (add c) <<op1,op2>> = add_data op1 op2

effect (A_alu a) = alueffect a
delay (A_alu a) = aludelay a
component (A_alu a) = alucomp a
alueffect a s c = S_alu (aluout a (aluinput s c))
getaluout0 (S_alu y) = y
is_proper_alu (S_alu y) = good y
||clio: sy is_proper_alu extend


||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class inc.

inc :: type
type inc_localstate = (data)
LOCAL_STATE ::= S_inc !inc_localstate |+
ACTION ::= A_inc !inc_ACT |+
inc_ACT ::=
 increment !inc

incdelay (increment c) = #1
inccomp (increment c) = C_inc c
incout (increment c) in = inc_data in

effect (A_inc a) = inceffect a
delay (A_inc a) = incdelay a
component (A_inc a) = inccomp a
inceffect a s c = S_inc (incout a (incinput s c))
getincout0 (S_inc y) = y
is_proper_inc (S_inc y) = good y
||clio: sy is_proper_inc extend


||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class matcher.

matcher :: type
type matcher_localstate = <<<<BOOL,BOOL>>,<<BOOL,BOOL>>>>
LOCAL_STATE ::= S_matcher !matcher_localstate |+
ACTION ::= A_matcher !matcher_ACT |+
matcher_ACT ::=
 match !matcher

matcherdelay (match c) = #1
matchercomp (match c) = C_matcher c
matcherout (match c) s <<src1,src2,dst>> =
  <<src1 = data_to_regaddr dst, data_to_regaddr src2 = data_to_regaddr dst>>
```

```
matcherstate (match c) s <<src1,src2,dst>> =
      <<src1=data_to_regaddr dst,data_to_regaddr src2 = data_to_regaddr dst>>

effect (A_matcher a) = matchereffect a
delay (A_matcher a) = matcherdelay a
component (A_matcher a) = matchercomp a
matchereffect a s c =
 S_matcher <<matcherstate a (getmatcherstate (current s c))
                                                    (matcherinput s c),
            matcherout a (getmatcherstate (current s c)) (matcherinput s c)>>
getmatcherstate (S_matcher <<x,y>>) = x
getmatcherout0 (S_matcher <<x,<<y0,y1>> >>)  = y0
getmatcherout1 (S_matcher <<x,<<y0,y1>> >>)  = y1
is_proper_matcher (S_matcher <<x,<<y0,y1>> >>) = (! y0) & (! y1)
||clio: sy is_proper_matcher extend

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class bitlatch.

bitlatch :: type
type bitlatch_localstate = <<BOOL,(BOOL)>>
LOCAL_STATE ::= S_bitlatch !bitlatch_localstate |+
ACTION ::= A_bitlatch !bitlatch_ACT |+
bitlatch_ACT ::=
 setb !bitlatch

bitlatchdelay (setb c) = #1
bitlatchcomp (setb c) = C_bitlatch c
bitlatchout (setb c) s in = in
bitlatchstate (setb c) s in = in

effect (A_bitlatch a) = bitlatcheffect a
delay (A_bitlatch a) = bitlatchdelay a
component (A_bitlatch a) = bitlatchcomp a
bitlatcheffect a s c =
 S_bitlatch <<bitlatchstate a (getbitlatchstate (current s c))
                                                    (bitlatchinput s c),
             bitlatchout a (getbitlatchstate (current s c))
                                                     (bitlatchinput s c)>>
getbitlatchstate (S_bitlatch <<x,y>>) = x
getbitlatchout0 (S_bitlatch <<x,y>>) = y
is_proper_bitlatch (S_bitlatch <<x,y>>) = ! y
||clio: sy is_proper_bitlatch extend
Goodbitlatch 's' 'out':== 's'='out'

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class special_regfile.

special_regfile :: type
type special_regfile_localstate = <<sregaddr->data,<<(byte),(data)>>>>
LOCAL_STATE ::= S_special_regfile !special_regfile_localstate |+
ACTION ::= A_special_regfile !special_regfile_ACT |+
special_regfile_ACT ::=
 sunload !special_regfile |
 sread !special_regfile |
 sload !special_regfile

special_regfiledelay (sunload c) = #1
special_regfilecomp (sunload c) = C_special_regfile c
special_regfileout (sunload c) s <<dst,din,v1,v2,v3,status,b,src>> =
                                          <<getbyte b (s PVT), s src>>
special_regfilestate (sunload c) s <<dst,din,v1,v2,v3,status,b,src>> = s

special_regfiledelay (sread c) = #1
```

```
special_regfilecomp (sread c) = C_special_regfile c
special_regfileout (sread c) s <<dst,din,v1,v2,v3,status,b,src>> =
                                          <<getbyte b (s PVT),  s dst>>
special_regfilestate (sread c) s <<dst,din,v1,v2,v3,status,b,src>> =
                                        vtrs_update s b v1 v2 v3 status

special_regfiledelay (sload c) = #1
special_regfilecomp (sload c) = C_special_regfile c
special_regfileout (sload c) s <<dst,din,v1,v2,v3,status,b,src>> =
  <<(protected dst -> getbyte b (s PVT); getbyte b (update s dst din PVT)),
    bottom>>
special_regfilestate (sload c) s <<dst,din,v1,v2,v3,status,b,src>> =
                                    ((protected dst)->s; update s dst din)

effect (A_special_regfile a) = special_regfileeffect a
delay (A_special_regfile a) = special_regfiledelay a
component (A_special_regfile a) = special_regfilecomp a
special_regfileeffect a s c =
 S_special_regfile <<special_regfilestate a (getspecial_regfilestate
                              (current s c)) (special_regfileinput s c),
                    special_regfileout a (getspecial_regfilestate
                                (current s c)) (special_regfileinput s c)>>
getspecial_regfilestate (S_special_regfile <<x,y>>) = x
getspecial_regfileout0 (S_special_regfile <<x,<<y0,y1>> >>)  = y0
getspecial_regfileout1 (S_special_regfile <<x,<<y0,y1>> >>)  = y1
is_proper_special_regfile (S_special_regfile <<x,<<y0,y1>> >>) =
                                                    (! y0) & (! y1)
||clio: sy is_proper_special_regfile extend


||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class mux4.

mux4 :: type
type mux4_localstate = (data)
LOCAL_STATE ::= S_mux4 !mux4_localstate |+
ACTION ::= A_mux4 !mux4_ACT |+
mux4_ACT ::=
 choose4_4 !mux4 |
 choose4_3 !mux4 |
 choose4_2 !mux4 |
 choose4_1 !mux4

mux4delay (choose4_4 c) = #0
mux4comp (choose4_4 c) = C_mux4 c
mux4out (choose4_4 c) <<in1,in2,in3,in4>> = in4

mux4delay (choose4_3 c) = #0
mux4comp (choose4_3 c) = C_mux4 c
mux4out (choose4_3 c) <<in1,in2,in3,in4>> = in3

mux4delay (choose4_2 c) = #0
mux4comp (choose4_2 c) = C_mux4 c
mux4out (choose4_2 c) <<in1,in2,in3,in4>> = in2

mux4delay (choose4_1 c) = #0
mux4comp (choose4_1 c) = C_mux4 c
mux4out (choose4_1 c) <<in1,in2,in3,in4>> = in1

effect (A_mux4 a) = mux4effect a
delay (A_mux4 a) = mux4delay a
component (A_mux4 a) = mux4comp a
mux4effect a s c = S_mux4 (mux4out a (mux4input s c))
getmux4out0 (S_mux4 y) = y
```

```
is_proper_mux4 (S_mux4 y) = ! y
||clio: sy is_proper_mux4 extend

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class byte_counter.

byte_counter :: type
type byte_counter_localstate = <<byte_num,(byte_num)>>
LOCAL_STATE ::= S_byte_counter !byte_counter_localstate |+
ACTION ::= A_byte_counter !byte_counter_ACT |+
byte_counter_ACT ::=
 reset_count !byte_counter |
 inc_byte !byte_counter

byte_counterdelay (reset_count c) = #0
byte_countercomp (reset_count c) = C_byte_counter c
byte_counterout (reset_count c) s in = BYTE0
byte_counterstate (reset_count c) s in = BYTE0

byte_counterdelay (inc_byte c) = #0
byte_countercomp (inc_byte c) = C_byte_counter c
byte_counterout (inc_byte c) s in = byte_inc s
byte_counterstate (inc_byte c) s in = byte_inc s

effect (A_byte_counter a) = byte_countereffect a
delay (A_byte_counter a) = byte_counterdelay a
component (A_byte_counter a) = byte_countercomp a
byte_countereffect a s c =
  S_byte_counter <<byte_counterstate a (getbyte_counterstate
                                (current s c)) (byte_counterinput s c),
                  byte_counterout a (getbyte_counterstate (current s c))
                                                  (byte_counterinput s c)>>
getbyte_counterstate (S_byte_counter <<x,y>>) = x
getbyte_counterout0 (S_byte_counter <<x,y>>) = y
is_proper_byte_counter (S_byte_counter <<x,y>>) = ! y
||clio: sy is_proper_byte_counter extend


ACTION ::= |
LOCAL_STATE ::= |
COMP ::=
 C_latch !latch |
 C_mux !mux |
 C_mem !mem |
 C_regfile !regfile |
 C_decoder !decoder |
 C_alu !alu |
 C_inc !inc |
 C_matcher !matcher |
 C_bitlatch !bitlatch |
 C_special_regfile !special_regfile |
 C_mux4 !mux4 |
 C_byte_counter !byte_counter

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Components (other than Controller and External)
|| and their connections.

mem ::= MEM |+
meminput s (C_mem MEM) =
  <<data_to_addr (getmux4out0 (current s (C_mux4 ADDR))),
    getmuxout0 (current s (C_mux DAT))>>
mux ::= DAT |+
muxinput s (C_mux DAT) =
```

```
   <<getmemout0 (current s (C_mem MEM)), bottom,
     getregfileout0 (current s (C_regfile REG))>>
latch ::= IREG |+
latchinput s (C_latch IREG) = getmuxout0 (current s (C_mux DAT))
mux4 ::= ADDR |+
mux4input s (C_mux4 ADDR) =
   <<getincout0 (current s (C_inc INC)),
     getlatchout0 (current s (C_latch HAND)),
     getlatchout0 (current s (C_latch IREG)),
     getlatchout0 (current s (C_latch RSLT))>>
latch ::= NXPC |+
latchinput s (C_latch NXPC) = getmux4out0 (current s (C_mux4 ADDR))
inc ::= INC |+
incinput s (C_inc INC) = getlatchout0 (current s (C_latch NXPC))
latch ::= HAND |+
latchinput s (C_latch HAND) =
              reset_to_addr (external_input_reset (current_input s))
bitlatch ::= RESET |+
bitlatchinput s (C_bitlatch RESET) = external_input_reset (current_input s)
latch ::= INST |+
latchinput s (C_latch INST) = getlatchout0 (current s (C_latch IREG))
latch ::= RSLT |+
latchinput s (C_latch RSLT) = getmuxout0 (current s (C_mux RS))
mux ::= RS |+
muxinput s (C_mux RS) =
   <<bottom, getaluout0 (current s (C_alu ALU)),
     getlatchout0 (current s (C_latch SOUT))>>
alu ::= ALU |+
aluinput s (C_alu ALU) =
   <<getmuxout0 (current s (C_mux OP1)),
     getmuxout0 (current s (C_mux OP2))>>
mux ::= OP1 |+
muxinput s (C_mux OP1) =
   <<getlatchout0 (current s (C_latch RSLT)), bottom,
     getregfileout0 (current s (C_regfile REG))>>
mux ::= OP2 |+
muxinput s (C_mux OP2) =
   <<getdecoderout4 (current s (C_decoder DEC)),
     getregfileout1 (current s (C_regfile REG)),
     getlatchout0 (current s (C_latch RSLT))>>
regfile ::= REG |+
regfileinput s (C_regfile REG) =
   <<getdecoderout3 (current s (C_decoder DEC)),
     data_to_regaddr (getdecoderout4 (current s (C_decoder DEC))),
     data_to_regaddr (getlatchout0 (current s (C_latch DST))),
     getmuxout0 (current s (C_mux DVAL))>>
latch ::= DST |+
latchinput s (C_latch DST) = getdecoderout2 (current s (C_decoder DEC))
decoder ::= DEC |+
decoderinput s (C_decoder DEC) = getlatchout0 (current s (C_latch IREG))
matcher ::= MTCH |+
matcherinput s (C_matcher MTCH) =
    <<getdecoderout3 (current s (C_decoder DEC)),
      getdecoderout4 (current s (C_decoder DEC)),
      getlatchout0 (current s (C_latch DST))>>
mux ::= DVAL |+
muxinput s (C_mux DVAL) =
    <<getmuxout0 (current s (C_mux DAT)),
      getlatchout0 (current s (C_latch RSLT)), bottom>>
special_regfile ::= SREG |+
special_regfileinput s (C_special_regfile SREG) =
   <<data_to_sregaddr (getlatchout0 (current s (C_latch DST))),
```

```
     getlatchout0 (current s (C_latch RSLT)),
     external_input_v1 (current_input s),
     external_input_v2 (current_input s),
     external_input_v3 (current_input s),
     external_input_ST (current_input s),
     getbyte_counterout0 (current s (C_byte_counter BC)),
     regaddr_to_sregaddr (getdecoderout3 (current s (C_decoder DEC)))>>
byte_counter ::= BC |+
byte_counterinput s (C_byte_counter BC) = bottom
latch ::= SOUT |+
latchinput s (C_latch SOUT) =
          getspecial_regfileout1 (current s (C_special_regfile SREG))

latch ::= |
mux ::= |
mem ::= |
regfile ::= |
decoder ::= |
alu ::= |
inc ::= |
matcher ::= |
bitlatch ::= |
special_regfile ::= |
mux4 ::= |
byte_counter ::= |
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Output function
Out s = (getspecial_regfileout0 (current s (C_special_regfile SREG)))
external_output_PVT x0 = x0
inlist_from <<s,p,in>> = input_phases_of in

input_phases_of in = (map in input_phases) ++
                     input_phases_of (in. (#+ #num_phases))

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The scheduler.

scheduler WBA <<reset, inst, op, ind, eq1, eq2, cc>> 0 =
      (DECODE)++(SET_H)++(UNLOAD_INC)++[A_byte_counter(inc_byte BC)]
scheduler WBJ <<reset, inst, op, ind, eq1, eq2, cc>> 0 =
      (DECODE)++(SET_H)++(UNLOAD_INC)++[A_byte_counter(inc_byte BC)]
scheduler WBS <<reset, inst, op, ind, eq1, eq2, cc>> 0 =
      (DECODE)++(SET_H)++[A_regfile(d_unload REG),
       A_byte_counter(inc_byte BC)]
scheduler WBL <<reset, inst, op, ind, eq1, eq2, cc>> 0 =
      (DECODE)++(SET_H)++(READ_RESULT)++[A_byte_counter(inc_byte BC)]
scheduler WBJ <<reset, inst, op, ind, eq1, eq2, cc>> 1 =
      (JFETCH reset inst cc)++(GET_OPS ind)++(ALUOP op)++
                  [A_latch(set SOUT), A_special_regfile(sread SREG)]
scheduler WBJ <<reset, inst, op, ind, eq1, eq2, cc>> 2 =
      [A_latch(set NXPC), A_latch(set INST), A_byte_counter(inc_byte BC)]
scheduler WBJ <<reset, inst, op, ind, eq1, eq2, cc>> 3 =
      (PHASE3 op)++[Advance_controller]

scheduler WBS <<reset, inst, op, ind, eq1, eq2, cc>> 1 =
      (WRITE)++[A_special_regfile(sread SREG)]
scheduler WBA <<reset, inst, op, ind, eq1, eq2, cc>> 1 =
      (FETCH reset)++(FORWARD_OPS ind eq1 eq2 inst)++(ALUOP op)++
                        [A_latch(set SOUT), A_special_regfile(sread SREG)]
scheduler INTF <<reset, inst, op, ind, eq1, eq2, cc>> 1 =
                               (READ_IREG)++[A_special_regfile(sread SREG)]
scheduler INTC <<reset, inst, op, ind, eq1, eq2, cc>> 1 =
```

```
      (READ_INC)++(GET_OPS ind)++(ALUOP op)++[A_latch(set SOUT),
                                              A_special_regfile(sread SREG)]
 scheduler DF <<reset, inst, op, ind, eq1, eq2, cc>> 1 =
      (FETCH reset)++(GET_OPS ind)++(ALUOP op)++[A_latch(set SOUT),
       A_special_regfile(sread SREG)]
 scheduler WBL <<reset, inst, op, ind, eq1, eq2, cc>> 1 =
                                              [A_special_regfile(sread SREG)]
 scheduler INTF <<reset, inst, op, ind, eq1, eq2, cc>> 2 =
                            [A_latch(set NXPC), A_byte_counter(inc_byte BC)]
 scheduler INTF <<reset, inst, op, ind, eq1, eq2, cc>> 3 =
     [A_mux(choose1 DAT), A_latch(set IREG),
      A_special_regfile(sread SREG)]++[Advance_controller]

 scheduler INTF <<reset, inst, op, ind, eq1, eq2, cc>> 0 =
                                                [A_byte_counter(inc_byte BC)]
 scheduler INTC <<reset, inst, op, ind, eq1, eq2, cc>> 3 =
                                           (PHASE3 op)++[Advance_controller]

 scheduler DFI <<reset, inst, op, ind, eq1, eq2, cc>> 3 =
                                           (PHASE3 op)++[Advance_controller]

 scheduler INTC <<reset, inst, op, ind, eq1, eq2, cc>> 0 =
             (DECODE)++(SET_H)++(UNLOAD_INC)++[A_byte_counter(inc_byte BC)]
 scheduler INTC <<reset, inst, op, ind, eq1, eq2, cc>> 2 =
        [A_latch(set NXPC), A_latch(set INST), A_byte_counter(inc_byte BC)]
 scheduler DFI <<reset, inst, op, ind, eq1, eq2, cc>> 2 =
        [A_latch(set NXPC), A_latch(set INST), A_byte_counter(inc_byte BC)]
 scheduler DF <<reset, inst, op, ind, eq1, eq2, cc>> 3 =
                                           (PHASE3 op)++[Advance_controller]

 scheduler DF <<reset, inst, op, ind, eq1, eq2, cc>> 0 =
             (DECODE)++(SET_H)++(UNLOAD_INC)++[A_byte_counter(inc_byte BC)]
 scheduler WBA <<reset, inst, op, ind, eq1, eq2, cc>> 2 =
    [A_latch(set NXPC), A_latch(set INST)]++(LOAD_RESULT inst)++
                                              [A_byte_counter(inc_byte BC)]
 scheduler WBA <<reset, inst, op, ind, eq1, eq2, cc>> 3 =
                                           (PHASE3 op)++[Advance_controller]

 scheduler WBL <<reset, inst, op, ind, eq1, eq2, cc>> 2 =
             (LOAD_DATA)++[A_latch(set INST), A_byte_counter(inc_byte BC)]
 scheduler WBL <<reset, inst, op, ind, eq1, eq2, cc>> 3 =
                      [A_special_regfile(sread SREG)]++[Advance_controller]

 scheduler DF <<reset, inst, op, ind, eq1, eq2, cc>> 2 =
        [A_latch(set NXPC), A_latch(set INST), A_byte_counter(inc_byte BC)]
 scheduler WBS <<reset, inst, op, ind, eq1, eq2, cc>> 2 =
                           [A_latch(set INST), A_byte_counter(inc_byte BC)]
 scheduler WBS <<reset, inst, op, ind, eq1, eq2, cc>> 3 =
                      [A_special_regfile(sread SREG)]++[Advance_controller]

DECODE = [A_decoder(decode DEC), A_matcher(match MTCH)]
SET_H = [A_latch(set HAND), A_bitlatch(setb RESET)]
UNLOAD_INC = [A_regfile(unload REG), A_special_regfile(sunload SREG),
              A_inc(increment INC)]
scheduler DFI <<reset, inst, op, ind, eq1, eq2, cc>> 0 =
                      (DECODE)++(UNLOAD_INC)++[A_byte_counter(inc_byte BC)]
READ_RESULT = [A_mux4(choose4_4 ADDR), A_mem(read MEM)]
GET_OPS ind = [A_mux(choose3 OP1)]++((ind)->[A_mux(choose2 OP2)];
                                                [A_mux(choose1 OP2)])
scheduler DFI <<reset, inst, op, ind, eq1, eq2, cc>> 1 =
  (READ_HAND)++(GET_OPS ind)++(ALUOP op)++
                           [A_latch(set SOUT), A_special_regfile(sread SREG)]
ALUOP op = ((op=CNE)->[A_alu(compare ALU)];[A_alu(add ALU)])
```

```
READ_HAND = [A_mux4(choose4_2 ADDR), A_mem(read MEM)]
READ_INC = [A_mux4(choose4_1 ADDR), A_mem(read MEM)]
FETCH reset = ((reset)->(READ_HAND);(READ_INC))
READ_IREG = [A_mux4(choose4_3 ADDR), A_mem(read MEM)]
WRITE = [A_mux4(choose4_4 ADDR), A_mux(choose3 DAT), A_mem(write MEM)]
FORWARD_OPS ind eq1 eq2 inst =
  ((eq1 & (inst ~= SADD))->[A_mux(choose1 OP1)];[A_mux(choose3 OP1)])++
  ((ind)->((eq2 & (inst ~= SADD))->[A_mux(choose3 OP2)];
                                [A_mux(choose2 OP2)]);[A_mux(choose1 OP2)])
JFETCH reset inst cc =
  ((reset)->(READ_HAND);(((inst=JMP)xor((inst=JIT)&cc)xor((inst=JIF)&
                                          (~cc)))->(READ_RESULT);(READ_INC)))
LOAD_DATA = [A_mux(choose1 DAT), A_mux(choose1 DVAL), A_regfile(load REG)]
LOAD_RESULT inst = [A_mux(choose2 DVAL)]++
       ((inst=SADD)->[A_special_regfile(sload SREG)];[A_regfile(load REG)])
PHASE3 op =
  [A_mux(choose1 DAT), A_latch(set IREG),
   A_latch(set DST)]++(SET_RESULT op)++[A_special_regfile(sread SREG)]
SET_RESULT op = ((op=MOVE)->[A_mux(choose3 RS)];[A_mux(choose2 RS)])++
                                                        [A_latch(set RSLT)]
scheduler s in 3 = [Advance_controller]
scheduler s in t = []

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Proper state predicates.

is_proper_state `s` :==
`is_proper_contr (current s Controller)` = `True`
& `is_proper_mem (current s (C_mem MEM))` = `True`
& `is_proper_mux (current s (C_mux DAT))` = `True`
& `is_proper_latch (current s (C_latch IREG))` = `True`
& `is_proper_mux4 (current s (C_mux4 ADDR))` = `True`
& `is_proper_latch (current s (C_latch NXPC))` = `True`
& `is_proper_inc (current s (C_inc INC))` = `True`
& `is_proper_latch (current s (C_latch HAND))` = `True`
& `is_proper_bitlatch (current s (C_bitlatch RESET))` = `True`
& `is_proper_latch (current s (C_latch INST))` = `True`
& `is_proper_latch (current s (C_latch RSLT))` = `True`
& `is_proper_mux (current s (C_mux RS))` = `True`
& `is_proper_alu (current s (C_alu ALU))` = `True`
& `is_proper_mux (current s (C_mux OP1))` = `True`
& `is_proper_mux (current s (C_mux OP2))` = `True`
& `is_proper_regfile (current s (C_regfile REG))` = `True`
& `is_proper_latch (current s (C_latch DST))` = `True`
& `is_proper_decoder (current s (C_decoder DEC))` = `True`
& `is_proper_matcher (current s (C_matcher MTCH))` = `True`
& `is_proper_mux (current s (C_mux DVAL))` = `True`
& `is_proper_special_regfile (current s
                                      (C_special_regfile SREG))` = `True`
& `is_proper_byte_counter (current s (C_byte_counter BC))` = `True`
& `is_proper_latch (current s (C_latch SOUT))` = `True`

Proper_state `s` :=
`pending_changes s`=`[]`
& is_proper_state `s`
& Goodmem `getmemstate(current s (C_mem MEM))`
                  `getmemout0(current s (C_mem MEM))`
& Goodlatch `getlatchstate(current s (C_latch IREG))`
                    `getlatchout0(current s (C_latch IREG))`
& Goodlatch `getlatchstate(current s (C_latch NXPC))`
                    `getlatchout0(current s (C_latch NXPC))`
```

```
& Goodlatch 'getlatchstate(current s (C_latch HAND))'
                    'getlatchout0(current s (C_latch HAND))'
& Goodbitlatch 'getbitlatchstate(current s (C_bitlatch RESET))'
                      'getbitlatchout0(current s (C_bitlatch RESET))'
& Goodlatch 'getlatchstate(current s (C_latch INST))''
                    'getlatchout0(current s (C_latch INST))'
& Goodlatch 'getlatchstate(current s (C_latch RSLT))'
                    'getlatchout0(current s (C_latch RSLT))'
& Goodregfile 'getregfilestate(current s (C_regfile REG))'
                     'getregfileout0(current s (C_regfile REG))'
                     'getregfileout1(current s (C_regfile REG))'
                     'getregfileout2(current s (C_regfile REG))'
& Goodlatch 'getlatchstate(current s (C_latch DST))'
                    'getlatchout0(current s (C_latch DST))'
& Goodlatch 'getlatchstate(current s (C_latch SOUT))'
                    'getlatchout0(current s (C_latch SOUT))'
& Proper_External 's'

||clio: symbol Proper_state extend_auto
||clio: symbol pending_changes extend_auto
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| For export to other modules, we define functions
|| access the internal state of each component that has one.
get_MEM_state s = getmemstate (current s (C_mem MEM))
get_IREG_state s = getlatchstate (current s (C_latch IREG))
get_NXPC_state s = getlatchstate (current s (C_latch NXPC))
get_HAND_state s = getlatchstate (current s (C_latch HAND))
get_RESET_state s = getbitlatchstate (current s (C_bitlatch RESET))
get_INST_state s = getlatchstate (current s (C_latch INST))
get_RSLT_state s = getlatchstate (current s (C_latch RSLT))
get_REG_state s = getregfilestate (current s (C_regfile REG))
get_DST_state s = getlatchstate (current s (C_latch DST))
get_MTCH_state s = getmatcherstate (current s (C_matcher MTCH))
get_SREG_state s =
          getspecial_regfilestate (current s (C_special_regfile SREG))
get_BC_state s = getbyte_counterstate (current s (C_byte_counter BC))
get_SOUT_state s = getlatchstate (current s (C_latch SOUT))


||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Auxillary definitions.

 ABS s = <<get_MEM_state s, get_NXPC_state s,
           get_REG_state s, get_INST_state s,
           get_SREG_state s,  time_abs(inlist_from s)>>

time_abs (<<r0,v10,v20,v30,st0>>:
            <<r1,v11,v21,v31,st1>>:<<r3,v13,v23,v33,st3>>:rest) =
         <<r0,[v11,v13],[v21,v23],[v31,v33],[st1,st3]>>: (time_abs rest)

vtrs_update s b v1 v2 v3 st =
     update(update(update(update s VT1 (set_byte b (s VT1) v1))
                                  VT2 (set_byte b (s VT2) v2))
                                  VT3 (set_byte b (s VT3) v3))
                                  STATUS (set_byte b (s STATUS) st)

mux4input = bottom

 || The following should be generated by the tool

path_taken s {controllerstate s = WBA} =
```

```
    (external_input_reset(current_input s))-> ACTPATH1;ACTPATH0
path_taken s {controllerstate s = WBS} =
    (external_input_reset(current_input s))-> ACTPATH4
    (external_input_reset(nth_input (#4) s))->ACTPATH3;ACTPATH2
path_taken s {controllerstate s = WBL} =
    (external_input_reset(current_input s))-> ACTPATH7
    (external_input_reset(nth_input (#4) s))->ACTPATH6;ACTPATH5
path_taken s {controllerstate s = WBJ} =
    (external_input_reset(current_input s))-> ACTPATH9;ACTPATH8


||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The cutpoints and loop conditions.

CUTPOINT ::= ACT

at_ACT s = ACT_state (controllerstate s)

ACT_state WBJ = True
ACT_state WBA = True
ACT_state WBS = True
ACT_state WBL = True
ACT_state x = False

PATH ::= ACTPATH0 |+
path_start ACTPATH0 = ACT
path_end ACTPATH0 = ACT
path_length ACTPATH0 = #1

path_condition `ACTPATH0` `s` :==
TRUE
& `controllerstate s`=`WBA`
& `(resetof (controllerinput (iterate Execute (#1) s)))`=`False`

PATH ::= ACTPATH1 |+
path_start ACTPATH1 = ACT
path_end ACTPATH1 = ACT
path_length ACTPATH1 = #3

path_condition `ACTPATH1` `s` :==
TRUE
& `controllerstate s`=`WBA`
& `(resetof (controllerinput (iterate Execute (#1) s)))`=`True`
& `controllerstate (iterate Execute (#1) s)`=`INTF`
& `controllerstate (iterate Execute (#2) s)`=`INTC`

PATH ::= ACTPATH2 |+
path_start ACTPATH2 = ACT
path_end ACTPATH2 = ACT
path_length ACTPATH2 = #2

path_condition `ACTPATH2` `s` :==
TRUE
& `controllerstate s`=`WBS`
& `(resetof (controllerinput (iterate Execute (#1) s)))`=`False`
& `controllerstate (iterate Execute (#1) s)`=`DF`
& `(resetof (controllerinput (iterate Execute (#2) s)))`=`False`

PATH ::= ACTPATH3 |+
path_start ACTPATH3 = ACT
path_end ACTPATH3 = ACT
path_length ACTPATH3 = #4

path_condition `ACTPATH3` `s` :==
TRUE
& `controllerstate s`=`WBS`
& `(resetof (controllerinput (iterate Execute (#1) s)))`=`False`
```

```
& `controllerstate (iterate Execute (#1) s)`=`DF`
& `(resetof (controllerinput (iterate Execute (#2) s)))`=`True`
& `controllerstate (iterate Execute (#2) s)`=`INTF`
& `controllerstate (iterate Execute (#3) s)`=`INTC`

PATH ::= ACTPATH4 |+
path_start ACTPATH4 = ACT
path_end ACTPATH4 = ACT
path_length ACTPATH4 = #4

path_condition `ACTPATH4` `s` :==
TRUE
& `controllerstate s`=`WBS`
& `(resetof (controllerinput (iterate Execute (#1) s)))`=`True`
& `controllerstate (iterate Execute (#1) s)`=`DFI`
& `controllerstate (iterate Execute (#2) s)`=`INTF`
& `controllerstate (iterate Execute (#3) s)`=`INTC`

PATH ::= ACTPATH5 |+
path_start ACTPATH5 = ACT
path_end ACTPATH5 = ACT
path_length ACTPATH5 = #2

path_condition `ACTPATH5` `s` :==
TRUE
& `controllerstate s`=`WBL`
& `(resetof (controllerinput (iterate Execute (#1) s)))`=`False`
& `controllerstate (iterate Execute (#1) s)`=`DF`
& `(resetof (controllerinput (iterate Execute (#2) s)))`=`False`

PATH ::= ACTPATH6 |+
path_start ACTPATH6 = ACT
path_end ACTPATH6 = ACT
path_length ACTPATH6 = #4

path_condition `ACTPATH6` `s` :==
TRUE
& `controllerstate s`=`WBL`
& `(resetof (controllerinput (iterate Execute (#1) s)))`=`False`
& `controllerstate (iterate Execute (#1) s)`=`DF`
& `(resetof (controllerinput (iterate Execute (#2) s)))`=`True`
& `controllerstate (iterate Execute (#2) s)`=`INTF`
& `controllerstate (iterate Execute (#3) s)`=`INTC`

PATH ::= ACTPATH7 |+
path_start ACTPATH7 = ACT
path_end ACTPATH7 = ACT
path_length ACTPATH7 = #4

path_condition `ACTPATH7` `s` :==
TRUE
& `controllerstate s`=`WBL`
& `(resetof (controllerinput (iterate Execute (#1) s)))`=`True`
& `controllerstate (iterate Execute (#1) s)`=`DFI`
& `controllerstate (iterate Execute (#2) s)`=`INTF`
& `controllerstate (iterate Execute (#3) s)`=`INTC`

PATH ::= ACTPATH8 |+
path_start ACTPATH8 = ACT
path_end ACTPATH8 = ACT
path_length ACTPATH8 = #1

path_condition `ACTPATH8` `s` :==
TRUE
& `controllerstate s`=`WBJ`
& `(resetof (controllerinput (iterate Execute (#1) s)))`=`False`

PATH ::= ACTPATH9 |+
path_start ACTPATH9 = ACT
```

```
path_end ACTPATH9 = ACT
path_length ACTPATH9 = #3

path_condition 'ACTPATH9' 's' :==
TRUE
& 'controllerstate s'='WBJ'
& '(resetof (controllerinput (iterate Execute (#1) s)))'='True'
& 'controllerstate (iterate Execute (#1) s)'='INTF'
& 'controllerstate (iterate Execute (#2) s)'='INTC'


ACT_Invariant 's' :==
                'prefetch(ABS s)'='get_IREG_state s'
                 & 'OP_switch (current_opclass(ABS s))'= 'controllerstate s'
                 & 'current_dst (ABS s)'='data_to_regaddr(get_DST_state s)'
                 & 'current_result(ABS s)' = 'get_RSLT_state s'


Invariant 'ACT' 's' := ACT_Invariant 's'

Invariant 'p' 's' := TRUE
PATH ::= |


ACT_ACT_Advance_Relation 's1' 's2':==   'ABS s2'='Step (ABS s1)'

Advance_Relation 'ACT' 'ACT' 's1' 's2' := ACT_ACT_Advance_Relation 's1' 's2'

Advance_Relation 'p1' 'p2' 's1' 's2' := TRUE
path_precond 'path' 's' :=
  '!path'='True'
  & path_condition 'path' 's'
  & Invariant 'path_start path' 's'
||clio: sy path_precond extend
||clio: sy Invariant extend

path_postcond 'path' 's' :=
   Invariant 'path_end path' 'iterate Execute (path_length path) s'
   & Advance_Relation 'path_start path' 'path_end path'
                      's' 'iterate Execute (path_length path) s'

VC 'path' 's':=
 path_precond 'path' 's' => path_postcond 'path' 's'

VC_ok 's' :==  (path) VC 'path' 's'
VC_ok_lemma :==  Proper_state 's' => VC_ok 's'
Timing_ok 's' :==
   Proper_state 'Execute s'
Timing_ok_lemma :==
   Timing_ok 's', Proper_state 's'

Timing_ok_case 's':==
   Timing_ok 's', 'controllerstate s'='c'

Timing_ok_by_cases :== Proper_state 's' => Timing_ok_case 's'
```

4.3 Common Part

This section contains the definitions of the types and functions that are used in the abstract as well as the design specifications of FtCayuga.

```
FROM CommonTheorySec IMPORT data,byte,addr,regaddr,sregaddr,opcode,opclass,
```

```
                        inc_data,decr_data, add_data, sub_data,cne_data,
                        data_to_addr, data_to_regaddr,
                        data_to_sregaddr, regaddr_to_sregaddr, DATUM
                        opcodeof,indirect,src1of,src2of,dstof,opclassof,
                        R0,R1,R2,R3,R4,nodata,dataerror, protected
                        JIF,JIT,JMP,ADD,LD,ST,CNE,MOVE,SADD
                        update,LSB,ACLASS,SCLASS,LCLASS,JCLASS, noop,
                        error, JUMP_ERROR, STORE_ERROR, NO_ERROR, addr_0

|| The abstract state consists of the memory state, the NXPC, the state of
|| the registers, the current instruction, and an input stream.
type ABS_EXTSTATE = <<BOOL, [byte], [byte], [byte], [byte]>>
||type ABS_STATE = <<addr->data, data, regaddr->data, data,
                                                sregaddr->data,[EXTSTATE]>>
mem <<m, p, r, i, sr,in>> = m
nxpc <<m, p, r, i, sr,in>> = p
reg <<m, p, r, i, sr,in>> = r
instrn <<m, p, r, i, sr,in>> = i
ins <<m, p, r, i, sr,in>> = in
sreg <<m, p, r, i, sr,in>> = sr

|| Step has type ABS_STATE->ABS_STATE.

Step s = <<newmem s, newnxpc s , newreg s, newinstr s , newsreg s,newins s>>


newmem s = store_effect s , current_opclass s = SCLASS
           mem s

newreg s  = alu_effect s, current_opclass s = ACLASS
            load_effect s, current_opclass s = LCLASS
            reg s

|| The new current instruction is usually fetched as follows:
prefetch s = mem s (data_to_addr(nxpc s))

|| But when there is an interrupt the instruction isn't prefetched.
|| Right now, there's only one interrupt, so the handler is at whatever
|| is in memory at addr_0.
handler s = newmem s addr_0

handler_fetch s = newmem s (data_to_addr(handler s))

newinstr s = handler_fetch s , interrupt s
             prefetch s

|| The new next pc.
newnxpc s =  inc_data(handler s), interrupt s
             jump_effect s, current_opclass s = JCLASS
             inc_data(nxpc s)


|| We must define at the abstract level the function that tells us
|| how many inputs to take from the input stream.

load_or_store x = (x=LCLASS)xor(x=SCLASS)

interruptof <<int,v1,v2,v3,st>> = int
interrupt s { load_or_store (opclassof (opcodeof (instrn s)))}
     = (interruptof(hd (ins s))|interruptof(hd(tl (ins s))))
interrupt s = interruptof(hd (ins s))

num_cycles s { load_or_store (opclassof (opcodeof (instrn s)))}
     = (interrupt s) ->#4;#2
num_cycles s = (interrupt s) ->#3;#1
```

```
newins s = remove (num_cycles s) (ins s)

remove Zero l = l
remove (Succ n) l = remove n (tl l)


|| The current instruction is decoded:
current_op s = opcodeof (instrn s)
current_opclass s = opclassof (current_op s)
current_ind s = indirect (instrn s)
current_src1 s = src1of (instrn s)
current_src2 s = src2of (instrn s)
current_dst s = data_to_regaddr (dstof (instrn s))
current_sdst s = data_to_sregaddr (dstof (instrn s))

|| Operand1 is the contents of register src1.
current_operand1 s = reg s (current_src1 s)

|| If the addressing mode is indirect then operand2 is the contents of
|| the register src2, otherwise it's src2 itself.
current_operand2 s = reg s (data_to_regaddr (current_src2 s)), current_ind s
                     current_src2 s

|| The alu operation is usually add:
current_aluop s = which_op (current_op s)
which_op CNE = cne_data
which_op x = add_data

|| The current result is computed by appying the current alu operation
|| to the current operands, unless the opcode is MOVE, in which case
|| it's the contents of special-register src1.
current_result s {current_op s = MOVE} =
                                  sreg s (regaddr_to_sregaddr(current_src1 s))
current_result s = current_aluop s (current_operand1 s) (current_operand2 s)

alu_effect s = update (reg s) (current_dst s) (current_result s)

load_effect s = update (reg s) (current_dst s)
                               (mem s (data_to_addr(current_result s)))

store_effect s = update (mem s) (data_to_addr(current_result s))
                                (reg s (current_dst s))

jump_effect s = current_result s, jump_ok s
                inc_data(nxpc s)

jump_ok s = ((current_op s)=JMP)
            xor (((current_op s)=JIT) & (LSB(reg s (current_dst s))))
            xor (((current_op s)=JIF) & ~(LSB(reg s (current_dst s))))

|| A Special-Add instruction (SADD) stores its result in the
|| special register file, unless the dst is write-protected,
|| in which case it has no effect.
newsreg s = update (sreg s) (current_sdst s) (current_result s)
                        , (current_op s = SADD) & ~(protected((current_sdst s)))
            sreg s

|| Specification of power-up (which includes a reset)

reset_addr s = mem s addr_0
reset_inst s = mem s (data_to_addr (reset_addr s))
abs_reset s = <<mem s, inc_data (reset_addr s),
                reg s, reset_inst s, sreg s, remove (#3) (ins s) >>

||clio: mod * off
||clio: add *
```

```
||clio: mod * on
```

5 IcNet Specification

5.1 Abstract Specification

```
FROM CommonTheorySec IMPORT Word, DATUM, PVT, VT1,VT2,VT3,
                            BYTE0, BYTE1, BYTE2, BYTE3, getbyte
FROM VoterDesignSec IMPORT XNG11, XNG21, XNG31, CMPP, LDP1, LDP2, OUT1
FROM VoterAbstractSec IMPORT voterstep, cross_out, control, go_of, maj_of,
                             row1of, arrayof,
                             Proper_voterstate, next_voterstate,
                             compute_majority, to_proc, get_maj_val, maj_of
FROM FtCayugaAbstractSec IMPORT FtCayugaStep, vstartout, prvtout, bytecount,
                             Proper_ftcayuga, sregof, newPVT, newPVT2
FROM IcNetDesignSec IMPORT get_V1_state, get_V2_state, get_V3_state,
                           get_V4_state, get_FTC1_state, get_FTC2_state,
                           get_FTC3_state, get_FTC4_state, byzstep,
                           byz_cross1, byz_cross2, byz_cross3, byz_to_proc,
                           Execute, Proper_state, inlist_from, is_proper_ext
                           INDEX,ONE,TWO,THREE,FOUR,faulty, make_ftc_in,
                           byzgo, byzpvt1, byzpvt2, byzpvt3, byzCayugaStep

||*************************************************************************
|| Some generic functions we need.

Iterate Zero f x = x
Iterate (Succ n) f x = Iterate n f (f x)

select Zero (a:x) = a
select (Succ n) (a:x) = select n x
||*************************************************************************

|| The successor relation on the indicies.
succ ONE = TWO
succ TWO = THREE
succ THREE = FOUR
succ FOUR = ONE

succ2 = succ.succ
succ3 = succ2.succ

||********************** Fault modelling ******************************
|| Everything is defined in terms of the parameter "faulty" which is a
|| predicate on the type INDEX which tells us which fault regions are faulty.
|| The possible faults are listed in the following enumerated type:

FAULT ::= Region !INDEX | NO_FAULT

|| To model the fact that we are assuming at most one fault, we can suppose
|| that there is a constant, "the_fault" of type FAULT, and then define the
|| predicate "faulty" in terms of that constant.

the_fault :: FAULT
AXIOM '!the_fault'='True'

faulty index = (the_fault = (Region index))


||********************** Abstraction function *****************************
|| This abstraction function lets us view the state of the ic-net
```

```
|| as a function from indicies to ftcayuga states , a function from indicies
|| to voter states , and an input stream.
|| We wouldn't need this step if the spectool supported indexed components.
IcNetABS s = <<FTCStates s, VoterStates s, inlist_from s>>

FTCStates s ONE = get_FTC1_state s
FTCStates s TWO = get_FTC2_state s
FTCStates s THREE = get_FTC3_state s
FTCStates s FOUR = get_FTC4_state s

VoterStates s ONE = get_V1_state s
VoterStates s TWO = get_V2_state s
VoterStates s THREE = get_V3_state s
VoterStates s FOUR = get_V4_state s

||********************** Step function for IcNet ******************************
|| The abstract state change (behavior) of the voter-net.
|| What happens at each index depends on whether that index is faulty.
IcNetStep <<ftc,vtr, int:rest>> =
     <<newftc,newvtr ,rest>>
     where newftc index = fault_ftc_step index ftc (ftcinput index )
           newvtr index = fault_vtr_step index vtr (vtrinput index )
           ftcinput index = make_ftc_in (select_int index int)
                                        (fault_to_proc index ftc vtr)
           vtrinput index = Voterinput index ftc vtr


fault_ftc_step index s in =
    FtCayugaStep (s index) in , ~(faulty index)
    byzCayugaStep (s index) in

fault_vtr_step index s = voterstep (s index) , ~(faulty index)
                         byzstep (s index)

select_int ONE <<a,b,c,d>> = a
select_int TWO <<a,b,c,d>> = b
select_int THREE <<a,b,c,d>> = c
select_int FOUR <<a,b,c,d>> = d

fault_to_proc index ftc vtr =
    to_proc (vtr index) , ~(faulty index)
    byz_to_proc (vtr index)

Voterinput index ftc vtr =
    <<fault_from_proc index ftc,
      fault_cross THREE (succ3 index) vtr,
      fault_cross TWO (succ2 index) vtr,
      fault_cross ONE (succ index) vtr>>

|| The function "fault_from_proc" gives the input tuple of values coming
|| from the proccessors to the voter whose index is "index".

fault_from_proc index ftc =
    <<fault_vstart index ftc,
      fault_prvt (whichprvt index (succ index)) (succ index) ftc ,
      fault_prvt (whichprvt index (succ2 index)) (succ2 index) ftc,
      fault_prvt (whichprvt index (succ3 index)) (succ3 index) ftc>>

|| The expression "whichprvt i j" tells us which replica of the private value
|| of processor j is connected to voter i. This is only used when processor j
|| is faulty, and tells us which of byzpvt1, byzpvt2, and byzpvt3 was used.
whichprvt ONE i = 1
whichprvt TWO ONE = 1
whichprvt TWO i = 2
whichprvt THREE FOUR = 3
```

```
whichprvt THREE i = 2
whichprvt FOUR i = 3


fault_vstart i ftc = vstartout (ftc i), ~(faulty i)
                        byzgo (ftc i)

fault_prvt i1 i2 s = prvtout (s i2) , ~(faulty i2)
                     byz_pvt i1 (s i2)

fault_cross i1 i2 s = cross_out (s i2) , ~(faulty i2)
                      byz_cross i1 (s i2)

byz_cross ONE = byz_cross1
byz_cross TWO = byz_cross2
byz_cross THREE = byz_cross3

byz_pvt 1 = byzpvt1
byz_pvt 2 = byzpvt2
byz_pvt 3 = byzpvt3

IcNetStepLemma :=
    'IcNetABS (Execute s)'='IcNetStep (IcNetABS s)' , Proper_state 's'


|| What we have done with this abstraction step is to organize the
|| state of the ic-net as a function on indicies,
|| and to hide the cross connections.

||*************************************************************************
|| Here is the translation of the Proper_state predicate for the ic-net
|| to a predicate on abstract ic-net.
Proper_inlist 'l' := 'is_proper_ext (select n l)'='True', '!n'='True'

Proper_icnet '<<ftc,vtr,inlist>>' := Proper_voterstate 'vtr ONE'
                                   & Proper_voterstate 'vtr TWO'
                                   & Proper_voterstate 'vtr THREE'
                                   & Proper_voterstate 'vtr FOUR'
                                   & Proper_ftcayuga 'ftc ONE'
                                   & Proper_ftcayuga 'ftc TWO'
                                   & Proper_ftcayuga 'ftc THREE'
                                   & Proper_ftcayuga 'ftc FOUR'
                                   & Proper_inlist 'inlist'

|| We then prove :
ProperIcNetLemma :=
    Proper_state 's' => Proper_icnet 'IcNetABS s'
```

5.2 Design Specification

```
FROM CommonTheorySec IMPORT byte
FROM VoterAbstractSec IMPORT voterstep, cross_out, to_proc, VOTERSTATE
                             Proper_voterstate
FROM FtCayugaAbstractSec IMPORT FTCAYUGASTATE,FtCayugaStep, prvtout,
                                vstartout, Proper_ftcayuga, statusword

|| Generated by the spectool
||clio: symbol Execute never
||clio: mod * off
||clio: add *
||clio: mod * on
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
```

```
|| The Controller.

COMP ::= Controller |+
ACTION ::= Advance_controller|+
CONTROLSTATE :: type
LOCAL_STATE ::= S_Controller !CONTROLSTATE |+
getcontrolstate (S_Controller x) = x
is_proper_contr (S_Controller x) = !x
||clio: sy is_proper_contr extend
controllerstate s = getcontrolstate (current s Controller)
effect Advance_controller s c =
          S_Controller (nextstate (controllerstate s) (controllerinput s))
delay Advance_controller = #0
component Advance_controller = Controller

controllerinput s = bottom
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The Control states and the next_state function.


CONTROLSTATE ::= GO |+
nextstate GO in = GO
CONTROLSTATE ::= |
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The component External.

type EXTSTATE = <<BOOL, BOOL, BOOL, BOOL>>

external_input_int1 <<x0, x1, x2, x3>> = x0
external_input_int2 <<x0, x1, x2, x3>> = x1
external_input_int3 <<x0, x1, x2, x3>> = x2
external_input_int4 <<x0, x1, x2, x3>> = x3
is_proper_ext <<x0, x1, x2, x3>> = (!x0) & (!x1) & (!x2) & (!x3)
||clio: sy is_proper_ext extend

current_input <<s,p,in>> = in Zero
nth_input n <<s,p,in>> = in n
||clio: sy Proper_External extend
Proper_External '<<s,p,in>>' :=
(t::NAT) 'is_proper_ext (in t)'='True'
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| There are 2 clock phases per cycle.

num_phases = 2
input_phases  = [#1]

output_phase 0 = True
output_phase n = False

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The generic Execute function.

type SYSTEM_STATE = COMP->LOCAL_STATE
type INPUT_STREAM = NAT->EXTSTATE
type CHANGE = <<COMP,NAT,LOCAL_STATE>>
type STATE = <<SYSTEM_STATE, [CHANGE], INPUT_STREAM>>

pending_changes <<s,p,in>> = p

current <<s,p,in>> c = bottom, pending p c
                       s c

pending [] c = False
pending (<<c,t,v>>:rest) c {t > Zero} = True
```

```
pending (<<c2,t,v>>:rest) c = pending rest c

Execute s = do_phases 0 s


do_phases n s = update_state s , n = num_phases
                do_phases (n+1) (do_phase n s)

Output s = generate_output 0 s


generate_output n s {n=num_phases} = []
generate_output n s {output_phase n} =
                  Out(do_phase n s) : generate_output (n+1) (do_phase n s)
generate_output n s = generate_output (n+1) (do_phase n s)

||clio: modify_rule "do_phases" count 20000
||clio: symbol do_phases never

update_state <<s,p,in>> = do_changes p <<s,[],in>>

do_phase n <<s,p,in>> =
  advance_inputstream (do_actions (current_schedule s2 n) s2)
  where s2 = update_state <<s,p,in>>

current_schedule s n = scheduler (controllerstate s) (controllerinput s) n

do_change <<c,Zero,v>> <<s,p,in>> = <<update s c v, p, in>>
do_change <<c,Succ n,v>> <<s,p,in>> = <<s, <<c,n,v>>:p, in>>

do_action a s = do_change (change_of a s) s

advance_inputstream <<s,p,in>> = <<s,p,in.Succ>>

change_of a s = <<component a, delay a, effect a s (component a)>>

do_changes [] s = s
do_changes (<<c,t,v>>:rest) s = do_changes rest (do_change <<c,t,v>> s)
|| This is a trick to handle conditional actions correctly
do_actions :: [ACTION]->STATE->STATE
AXIOM (s)'do_actions [] s' = 's'
AXIOM (a)(rest)(s)'do_actions (a:rest) s' = 'do_actions rest (do_action a s)'
update s c v c2 = (c=c2)->v; s c2

foldl op s [] = s
foldl op s (a:rest) = foldl op (op a s) rest

map f [] = []
map f (a:x) = (f a): (map f x)
list_to Zero = []
list_to (Succ n) = list_to n ++ [n]
iterate f Zero s = s
iterate f (Succ n) s = iterate f n (f s)
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Component Classes.

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class voter.

voter :: type
type voter_localstate = <<VOTERSTATE,<<(TO_PROC),(fourbyte),
                                                  (fourbyte),(fourbyte)>>>>
LOCAL_STATE ::= S_voter !voter_localstate |+
ACTION ::= A_voter !voter_ACT |+
voter_ACT ::=
 byzchange !voter |
 byzout !voter |
```

```
 changestate !voter |
 output !voter

voterdelay (byzchange c) = #1
votercomp (byzchange c) = C_voter c
voterout (byzchange c) s <<from_proc,v1,v2,v3>> =
                <<byz_to_proc s , byz_cross1 s, byz_cross2 s, byz_cross3 s>>
voterstate (byzchange c) s <<from_proc,v1,v2,v3>> =
                                        byzstep s <<from_proc,v1,v2,v3>>

voterdelay (byzout c) = #1
votercomp (byzout c) = C_voter c
voterout (byzout c) s <<from_proc,v1,v2,v3>> =
              <<byz_to_proc s , byz_cross1 s, byz_cross2 s, byz_cross3 s>>
voterstate (byzout c) s <<from_proc,v1,v2,v3>> = s

voterdelay (changestate c) = #1
votercomp (changestate c) = C_voter c
voterout (changestate c) s <<from_proc,v1,v2,v3>> =
                    <<to_proc s , cross_out s, cross_out s, cross_out s >>
voterstate (changestate c) s <<from_proc,v1,v2,v3>> =
                                    voterstep s <<from_proc,v1,v2,v3>>

voterdelay (output c) = #1
votercomp (output c) = C_voter c
voterout (output c) s <<from_proc,v1,v2,v3>> =
                   <<to_proc s , cross_out s , cross_out s , cross_out s >>
voterstate (output c) s <<from_proc,v1,v2,v3>> = s

effect (A_voter a) = votereffect a
delay (A_voter a) = voterdelay a
component (A_voter a) = votercomp a
votereffect a s c =
   S_voter <<voterstate a (getvoterstate (current s c)) (voterinput s c),
            voterout a (getvoterstate (current s c)) (voterinput s c)>>
getvoterstate (S_voter <<x,y>>) = x
getvoterout0 (S_voter <<x,<<y0,y1,y2,y3>> >>)  = y0
getvoterout1 (S_voter <<x,<<y0,y1,y2,y3>> >>)  = y1
getvoterout2 (S_voter <<x,<<y0,y1,y2,y3>> >>)  = y2
getvoterout3 (S_voter <<x,<<y0,y1,y2,y3>> >>)  = y3
is_proper_voter (S_voter <<x,<<y0,y1,y2,y3>> >>) =
                                    (! y0) & (! y1) & (! y2) & (! y3)
||clio: sy is_proper_voter extend
Goodvoter 's' 'to_proc' 'cross1' 'cross2' 'cross3':== Proper_voterstate 's'

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The component class FtCayuga.

FtCayuga :: type
type FtCayuga_localstate =
          <<FTCAYUGASTATE,<<(fourbit),(fourbyte),(fourbyte),(fourbyte)>>>>
LOCAL_STATE ::= S_FtCayuga !FtCayuga_localstate |+
ACTION ::= A_FtCayuga !FtCayuga_ACT |+
FtCayuga_ACT ::=
 byzcayugaout !FtCayuga |
 cayugaout !FtCayuga |
 byzcayuga !FtCayuga |
 cayugastep !FtCayuga

FtCayugadelay (byzcayugaout c) = #1
FtCayugacomp (byzcayugaout c) = C_FtCayuga c
FtCayugaout (byzcayugaout c) s <<in,int>> =
                        <<(byzgo s), (byzpvt1 s), (byzpvt2 s), (byzpvt3 s)>>
FtCayugastate (byzcayugaout c) s <<in,int>> = s
```

```
FtCayugadelay (cayugaout c) = #1
FtCayugacomp (cayugaout c) = C_FtCayuga c
FtCayugaout (cayugaout c) s <<in,int>> =
                    <<(vstartout s), (prvtout s), (prvtout s), (prvtout s)>>
FtCayugastate (cayugaout c) s <<in,int>> = s

FtCayugadelay (byzcayuga c) = #1
FtCayugacomp (byzcayuga c) = C_FtCayuga c
FtCayugaout (byzcayuga c) s <<in,int>> =
                          <<(byzgo s), (byzpvt1 s), (byzpvt2 s), (byzpvt3 s)>>
FtCayugastate (byzcayuga c) s <<in,int>> =
                                          byzCayugaStep s (make_ftc_in int in)

FtCayugadelay (cayugastep c) = #1
FtCayugacomp (cayugastep c) = C_FtCayuga c
FtCayugaout (cayugastep c) s <<in,int>> =
                    <<(vstartout s), (prvtout s), (prvtout s), (prvtout s)>>
FtCayugastate (cayugastep c) s <<in,int>> =
                                          FtCayugaStep s (make_ftc_in int in)

effect (A_FtCayuga a) = FtCayugaeffect a
delay (A_FtCayuga a) = FtCayugadelay a
component (A_FtCayuga a) = FtCayugacomp a
FtCayugaeffect a s c =
  S_FtCayuga <<FtCayugastate a (getFtCayugastate (current s c))
                                                    (FtCayugainput s c),
  FtCayugaout a (getFtCayugastate (current s c)) (FtCayugainput s c)>>
getFtCayugastate (S_FtCayuga <<x,y>>) = x
getFtCayugaout0 (S_FtCayuga <<x,<<y0,y1,y2,y3>> >>)  = y0
getFtCayugaout1 (S_FtCayuga <<x,<<y0,y1,y2,y3>> >>)  = y1
getFtCayugaout2 (S_FtCayuga <<x,<<y0,y1,y2,y3>> >>)  = y2
getFtCayugaout3 (S_FtCayuga <<x,<<y0,y1,y2,y3>> >>)  = y3
is_proper_FtCayuga (S_FtCayuga <<x,<<y0,y1,y2,y3>> >>) =
                                          (! y0) & (! y1) & (! y2) & (! y3)
||clio: sy is_proper_FtCayuga extend
GoodFtCayuga 's' 'go' 'pvt1' 'pvt2' 'pvt3':== Proper_ftcayuga 's'

ACTION ::= |
LOCAL_STATE ::= |
COMP ::=
 C_voter !voter |
 C_FtCayuga !FtCayuga

!!*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Components (other than Controller and External)
|| and their connections.

FtCayuga ::= FTC1 |+
FtCayugainput s (C_FtCayuga FTC1) =
  <<getvoterout0 (current s (C_voter V1)),
    external_input_int1 (current_input s)>>
FtCayuga ::= FTC2 |+
FtCayugainput s (C_FtCayuga FTC2) =
  <<getvoterout0 (current s (C_voter V2)),
    external_input_int2 (current_input s)>>
FtCayuga ::= FTC3 |+
FtCayugainput s (C_FtCayuga FTC3) =
   <<getvoterout0 (current s (C_voter V3)),
     external_input_int3 (current_input s)>>
FtCayuga ::= FTC4 |+
```

```
FtCayugainput s (C_FtCayuga FTC4) = <<getvoterout0 (current s (C_voter V4)),
                                      external_input_int4 (current_input s)>>

voter ::= V1 |+
voterinput s (C_voter V1) =
 << <<getFtCayugaout0 (current s (C_FtCayuga FTC1)),
      getFtCayugaout1 (current s (C_FtCayuga FTC2)),
      getFtCayugaout1 (current s (C_FtCayuga FTC3)),
      getFtCayugaout1 (current s (C_FtCayuga FTC4))>>,
    getvoterout3 (current s (C_voter V4)),
    getvoterout2 (current s (C_voter V3)),
    getvoterout1 (current s (C_voter V2))>>
voter ::= V2 |+
voterinput s (C_voter V2) =
 << <<getFtCayugaout0 (current s (C_FtCayuga FTC2)),
      getFtCayugaout2 (current s (C_FtCayuga FTC3)),
      getFtCayugaout2 (current s (C_FtCayuga FTC4)),
      getFtCayugaout1 (current s (C_FtCayuga FTC1))>>,
    getvoterout3 (current s (C_voter V1)),
    getvoterout2 (current s (C_voter V4)),
    getvoterout1 (current s (C_voter V3))>>
voter ::= V3 |+
voterinput s (C_voter V3) =
  << <<getFtCayugaout0 (current s (C_FtCayuga FTC3)),
       getFtCayugaout3 (current s (C_FtCayuga FTC4)),
       getFtCayugaout2 (current s (C_FtCayuga FTC1)),
       getFtCayugaout2 (current s (C_FtCayuga FTC2))>>,
     getvoterout3 (current s (C_voter V2)),
     getvoterout2 (current s (C_voter V1)),
     getvoterout1 (current s (C_voter V4))>>
voter ::= V4 |+
voterinput s (C_voter V4) =
  << <<getFtCayugaout0 (current s (C_FtCayuga FTC4)),
       getFtCayugaout3 (current s (C_FtCayuga FTC1)),
       getFtCayugaout3 (current s (C_FtCayuga FTC2)),
       getFtCayugaout3 (current s (C_FtCayuga FTC3))>>,
     getvoterout3 (current s (C_voter V3)),
     getvoterout2 (current s (C_voter V2)),
     getvoterout1 (current s (C_voter V1))>>

voter ::= |
FtCayuga ::= |
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Output function
Out s = (bottom)
inlist_from <<s,p,in>> = input_phases_of in

input_phases_of in = (map in input_phases) ++
                     input_phases_of (in. (#+ #num_phases))

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| The scheduler.

scheduler G0 in 0 = (V1out)++(V2out)++(V3out)++(V4out)++
                                (FTC1out)++(FTC2out)++(FTC3out)++(FTC4out)
V1out = ((faulty ONE)->[A_voter(byzout V1)];[A_voter(output V1)])
V2out = ((faulty TWO)->[A_voter(byzout V2)];[A_voter(output V2)])
V3out = ((faulty THREE)->[A_voter(byzout V3)];[A_voter(output V3)])
V4out = ((faulty FOUR)->[A_voter(byzout V4)];[A_voter(output V4)])
V1state = ((faulty ONE)->[A_voter(byzchange V1)];[A_voter(changestate V1)])
V2state = ((faulty TWO)->[A_voter(byzchange V2)];[A_voter(changestate V2)])
V3state = ((faulty THREE)->[A_voter(byzchange V3)];[A_voter(changestate V3)])
```

```
V4state = ((faulty FOUR)->[A_voter(byzchange V4)];[A_voter(changestate V4)])
FTC1act = ((faulty ONE)->[A_FtCayuga(byzcayuga FTC1)];
                                                [A_FtCayuga(cayugastep FTC1)])
FTC2act = ((faulty TWO)->[A_FtCayuga(byzcayuga FTC2)];
                                                [A_FtCayuga(cayugastep FTC2)])
FTC3act = ((faulty THREE)->[A_FtCayuga(byzcayuga FTC3)];
                                                [A_FtCayuga(cayugastep FTC3)])
FTC4act = ((faulty FOUR)->[A_FtCayuga(byzcayuga FTC4)];
                                                [A_FtCayuga(cayugastep FTC4)])
scheduler GO in 1 = (V1state)++(V2state)++(V3state)++(V4state)++
          (FTC1act)++(FTC2act)++(FTC3act)++(FTC4act)++[Advance_controller]

FTC1out = ((faulty ONE)->[A_FtCayuga(byzcayugaout FTC1)];
                                                [A_FtCayuga(cayugaout FTC1)])
FTC2out = ((faulty TWO)->[A_FtCayuga(byzcayugaout FTC2)];
                                                 [A_FtCayuga(cayugaout FTC2)])
FTC3out = ((faulty THREE)->[A_FtCayuga(byzcayugaout FTC3)];
                                                 [A_FtCayuga(cayugaout FTC3)])
FTC4out = ((faulty FOUR)->[A_FtCayuga(byzcayugaout FTC4)];
                                                 [A_FtCayuga(cayugaout FTC4)])
scheduler s in 1 = [Advance_controller]
scheduler s in t = []

||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Proper state predicates.

is_proper_state `s` :==
`is_proper_contr (current s Controller)` = `True`
& `is_proper_FtCayuga (current s (C_FtCayuga FTC1))` = `True`
& `is_proper_FtCayuga (current s (C_FtCayuga FTC2))` = `True`
& `is_proper_FtCayuga (current s (C_FtCayuga FTC3))` = `True`
& `is_proper_FtCayuga (current s (C_FtCayuga FTC4))` = `True`
& `is_proper_voter (current s (C_voter V1))` = `True`
& `is_proper_voter (current s (C_voter V2))` = `True`
& `is_proper_voter (current s (C_voter V3))` = `True`
& `is_proper_voter (current s (C_voter V4))` = `True`

Proper_state `s` :=
`pending_changes s`=`[]`
& is_proper_state `s`
& GoodFtCayuga `getFtCayugastate(current s (C_FtCayuga FTC1))`
                        `getFtCayugaout0(current s (C_FtCayuga FTC1))`
                        `getFtCayugaout1(current s (C_FtCayuga FTC1))`
                        `getFtCayugaout2(current s (C_FtCayuga FTC1))`
                        `getFtCayugaout3(current s (C_FtCayuga FTC1))`
& GoodFtCayuga `getFtCayugastate(current s (C_FtCayuga FTC2))`
                        `getFtCayugaout0(current s (C_FtCayuga FTC2))`
                        `getFtCayugaout1(current s (C_FtCayuga FTC2))`
                        `getFtCayugaout2(current s (C_FtCayuga FTC2))`
                        `getFtCayugaout3(current s (C_FtCayuga FTC2))`
& GoodFtCayuga `getFtCayugastate(current s (C_FtCayuga FTC3))`
                        `getFtCayugaout0(current s (C_FtCayuga FTC3))`
                        `getFtCayugaout1(current s (C_FtCayuga FTC3))`
                        `getFtCayugaout2(current s (C_FtCayuga FTC3))`
                        `getFtCayugaout3(current s (C_FtCayuga FTC3))`
& GoodFtCayuga `getFtCayugastate(current s (C_FtCayuga FTC4))`
                        `getFtCayugaout0(current s (C_FtCayuga FTC4))`
                        `getFtCayugaout1(current s (C_FtCayuga FTC4))`
                        `getFtCayugaout2(current s (C_FtCayuga FTC4))`
                        `getFtCayugaout3(current s (C_FtCayuga FTC4))`
```

```
& Goodvoter `getvoterstate(current s (C_voter V1))`
                    `getvoterout0(current s (C_voter V1))`
                    `getvoterout1(current s (C_voter V1))`
                    `getvoterout2(current s (C_voter V1))`
                    `getvoterout3(current s (C_voter V1))`
& Goodvoter `getvoterstate(current s (C_voter V2))`
                    `getvoterout0(current s (C_voter V2))`
                    `getvoterout1(current s (C_voter V2))`
                    `getvoterout2(current s (C_voter V2))`
                    `getvoterout3(current s (C_voter V2))`
& Goodvoter `getvoterstate(current s (C_voter V3))`
                    `getvoterout0(current s (C_voter V3))`
                    `getvoterout1(current s (C_voter V3))`
                    `getvoterout2(current s (C_voter V3))`
                    `getvoterout3(current s (C_voter V3))`
& Goodvoter `getvoterstate(current s (C_voter V4))`
                    `getvoterout0(current s (C_voter V4))`
                    `getvoterout1(current s (C_voter V4))`
                    `getvoterout2(current s (C_voter V4))`
                    `getvoterout3(current s (C_voter V4))`
& Proper_External `s`

||clio: symbol Proper_state extend_auto
||clio: symbol pending_changes extend_auto
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| For export to other modules, we define functions
|| access the internal state of each component that has one.
get_FTC1_state s = getFtCayugastate (current s (C_FtCayuga FTC1))
get_FTC2_state s = getFtCayugastate (current s (C_FtCayuga FTC2))
get_FTC3_state s = getFtCayugastate (current s (C_FtCayuga FTC3))
get_FTC4_state s = getFtCayugastate (current s (C_FtCayuga FTC4))
get_V1_state s = getvoterstate (current s (C_voter V1))
get_V2_state s = getvoterstate (current s (C_voter V2))
get_V3_state s = getvoterstate (current s (C_voter V3))
get_V4_state s = getvoterstate (current s (C_voter V4))


||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--

|| Auxillary definitions.

make_ftc_in int <<v1,v2,v3,st,snd,fr>> =
    <<int,twolist v1, twolist v2, twolist v3, statusword st fr snd >>

twolist <<a,b>> = [a,b]
statusword :: *


  byzstep, byz_to_proc :: *
  byzc11, byzc12, byzc13, byzc14, byzc21, byzc22 :: *->byte
  byzc23, byzc24, byzc31, byzc32, byzc33, byzc34 :: *->byte
  byzCayugaStep :: *
  byzp11, byzp12, byzp13, byzp14, byzp21, byzp22 :: *->byte
  byzp23, byzp24, byzp31, byzp32, byzp33, byzp34 :: *->byte
  byzg1, byzg2, byzg3, byzg4 :: *->BOOL


  AXIOM `!(byzc11 x)`=`!x`
  AXIOM `!(byzc12 x)`=`!x`
  AXIOM `!(byzc13 x)`=`!x`
  AXIOM `!(byzc14 x)`=`!x`
```

```
   AXIOM '!(byzc21 x)'='!x'
   AXIOM '!(byzc22 x)'='!x'
   AXIOM '!(byzc23 x)'='!x'
   AXIOM '!(byzc24 x)'='!x'
   AXIOM '!(byzc31 x)'='!x'
   AXIOM '!(byzc32 x)'='!x'
   AXIOM '!(byzc33 x)'='!x'
   AXIOM '!(byzc34 x)'='!x'
   byz_cross1 x = <<byzc11 x, byzc12 x, byzc13 x, byzc14 x>>
   byz_cross2 x = <<byzc21 x, byzc22 x, byzc23 x, byzc24 x>>
   byz_cross3 x = <<byzc31 x, byzc32 x, byzc33 x, byzc34 x>>

   AXIOM '!(byzp11 x)'='!x'
   AXIOM '!(byzp12 x)'='!x'
   AXIOM '!(byzp13 x)'='!x'
   AXIOM '!(byzp14 x)'='!x'
   byzpvt1 x = <<byzp11 x, byzp12 x, byzp13 x, byzp14 x>>
   AXIOM '!(byzp21 x)'='!x'
   AXIOM '!(byzp22 x)'='!x'
   AXIOM '!(byzp23 x)'='!x'
   AXIOM '!(byzp24 x)'='!x'
   byzpvt2 x = <<byzp21 x, byzp22 x, byzp23 x, byzp24 x>>
   AXIOM '!(byzp31 x)'='!x'
   AXIOM '!(byzp32 x)'='!x'
   AXIOM '!(byzp33 x)'='!x'
   AXIOM '!(byzp34 x)'='!x'
   byzpvt3 x = <<byzp31 x, byzp32 x, byzp33 x, byzp34 x>>
   AXIOM '!(byzg1 x)'='!x'
   AXIOM '!(byzg2 x)'='!x'
   AXIOM '!(byzg3 x)'='!x'
   AXIOM '!(byzg4 x)'='!x'
   byzgo x = <<byzg1 x, byzg2 x, byzg3 x, byzg4 x>>

type FROM_PROC = <<proctuple,proctuple,proctuple,proctuple>>
type proctuple = <<byte,byte,byte,BOOL>>
type TO_PROC = <<twobyte,twobyte,twobyte,<<threebit,threebit>>,twobit,twobit>>
type fourbyte = <<byte,byte,byte,byte>>
type twobyte = <<byte,byte>>
type twobit = <<BOOL,BOOL>>
type threebit = <<BOOL,BOOL,BOOL>>
type fourbit = <<BOOL,BOOL,BOOL,BOOL>>

good_from_proc :: FROM_PROC->BOOL
good_from_proc <<a,b,c,d>> = (good_proctuple a) &
                             (good_proctuple b) &
                             (good_proctuple c) &
                             (good_proctuple d)
good_proctule :: proctuple->BOOL
good_proctuple <<a,b,c,d>> = !a & !b & !c & !d

||clio: symbol Proper_voterstate extend_auto
||clio: symbol Proper_ftcayuga extend_auto
||clio: symbol good_from_proc extend_auto
||clio: symbol good_proctuple extend_auto
is_proper_ext :: EXTSTATE->BOOL

INDEX ::= ONE | TWO | THREE | FOUR

|| Everything is defined in terms of this parameter.
faulty :: INDEX -> BOOL
```

```
||*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*--
|| The cutpoints and loop conditions.


PATH ::= |

Invariant 'p' 's' := TRUE

Advance_Relation 'p1' 'p2' 's1' 's2' := TRUE
path_start p = bottom
path_end p = bottom
path_length p = bottom

path_condition 'p' 's' :== FALSE

path_precond 'path' 's' :=
  '!path'='True'
  & path_condition 'path' 's'
  & Invariant 'path_start path' 's'
||clio: sy path_precond extend
||clio: sy Invariant extend

path_postcond 'path' 's' :=
   Invariant 'path_end path' 'iterate Execute (path_length path) s'
   & Advance_Relation 'path_start path' 'path_end path'
                      's' 'iterate Execute (path_length path) s'

VC 'path' 's':=
 path_precond 'path' 's' => path_postcond 'path' 's'

VC_ok 's' :==  (path) VC 'path' 's'
VC_ok_lemma :==  Proper_state 's' => VC_ok 's'
Timing_ok 's' :==
   Proper_state 'Execute s'
Timing_ok_lemma :==
   Timing_ok 's', Proper_state 's'

Timing_ok_case 's':==
   Timing_ok 's', 'controllerstate s'='c'

Timing_ok_by_cases :== Proper_state 's' => Timing_ok_case 's'
```

6 Common Theory

6.1 Type Definitions

```
|| Here's what we are assuming about the type number:
|| For ftcayuga, we must assume that a number consists of four bytes
byte_num ::= BYTE0|BYTE1|BYTE2|BYTE3
byte :: sort
number ::= Word byte byte byte byte
byte_inc BYTE0 = BYTE1
byte_inc BYTE1 = BYTE2
byte_inc BYTE2 = BYTE3
byte_inc BYTE3 = BYTE0
inc_num :: number->number
decr_num :: number->number
```

```
AXIOM '!(decr_num x)'='!x'
AXIOM '!(inc_num x)'='!x'
AXIOM 'decr_num (inc_num x)'='x'
add_num :: number->number->number
n0 :: number
sub_num :: number->number->number
AXIOM '!(add_num x y)'='!x & !y'
AXIOM '!(sub_num x y)'='!x & !y'
AXIOM 'add_num x n0'='x'
AXIOM 'add_num n0 x'='x'
n1 :: number
n2 :: number
n3 :: number

|| Here's what we are assuming about the type data:
data ::= nodata | dataerror | DATUM !number

inc_data (DATUM x) = DATUM (inc_num x)
inc_data x = x
decr_data (DATUM x) = DATUM (decr_num x)
decr_data x = x
add_data (DATUM x) (DATUM y) = DATUM (add_num x y)
sub_data (DATUM x) (DATUM y) = DATUM (sub_num x y)
cne_data x y = (x=y)->data_false;data_true
good (DATUM (Word x y z w)) = !x & !y & !z & !w
getbyte BYTE3 (DATUM (Word b3 b2 b1 b0)) = b3
getbyte BYTE2 (DATUM (Word b3 b2 b1 b0)) = b2
getbyte BYTE1 (DATUM (Word b3 b2 b1 b0)) = b1
getbyte BYTE0 (DATUM (Word b3 b2 b1 b0)) = b0
set_byte BYTE3 (DATUM (Word b3 b2 b1 b0)) v = DATUM (Word v b2 b1 b0)
set_byte BYTE2 (DATUM (Word b3 b2 b1 b0)) v = DATUM (Word b3 v b1 b0)
set_byte BYTE1 (DATUM (Word b3 b2 b1 b0)) v = DATUM (Word b3 b2 v b0)
set_byte BYTE0 (DATUM (Word b3 b2 b1 b0)) v = DATUM (Word b3 b2 b1 v)
get_byte = getbyte.num2byte
update_byte = set_byte.num2byte
num2byte 3 = BYTE3
num2byte 2 = BYTE2
num2byte 1 = BYTE1
num2byte 0 = BYTE0


LSB :: data->BOOL
AXIOM '!(LSB x)'='!x'
data_false :: data
data_true :: data
AXIOM 'good data_false'='True'
AXIOM 'good data_true'='True'
AXIOM 'LSB data_false'='False'
AXIOM 'LSB data_true'='True'

|| Memory addresses:
addr ::= ADDR !number
data_to_addr (DATUM x) = ADDR x
inc_addr (ADDR x) = ADDR (inc_num x)
addr_0 = ADDR n0
data_0 = DATUM n0

|| Here's what we need about the regaddr's:
regaddr ::= R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 |
            R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15 |
            R16 | R17 | R18 | R19 | R20 | R21 | R22 | R23 |
            R24 | R25 | R26 | R27 | R28 | R29 | R30 | R31

data_to_regaddr :: data -> regaddr
AXIOM '!(data_to_regaddr x)'='!x'
AXIOM 'data_to_regaddr (DATUM n0) '= 'R0'
AXIOM 'data_to_regaddr (DATUM n1) '= 'R1'
```

```
AXIOM 'data_to_regaddr (DATUM n2) '= 'R2'
AXIOM 'data_to_regaddr (DATUM n3) '= 'R3'

|| Special-reg-addrs
sregaddr ::= VT1|VT2|VT3|STATUS|PVT|+
data_to_sregaddr :: data->sregaddr
AXIOM '!(data_to_sregaddr x)'='!x'
regaddr_to_sregaddr :: regaddr->sregaddr
AXIOM '!(regaddr_to_sregaddr x)'='!x'
||Special registers VT1-VT3 and STATUS are write-protected
protected sr = (sr=VT1)xor(sr=VT2)xor(sr=VT3)xor(sr=STATUS)

|| The opcodes:
opcode ::= LD | ST | ADD | JMP | JIT | JIF | CNE | SADD | MOVE | ICOP
opclass ::= SCLASS | JCLASS | LCLASS | ACLASS | NONE

opcodeof :: data -> opcode
indirect :: data -> BOOL
dstof :: data -> data       || note: this is "padded" to be data
src1of :: data -> regaddr
src2of :: data -> data       || note: this is "padded" to be data
AXIOM '!(opcodeof x)'='!x'
AXIOM '!(indirect x)'='!x'
AXIOM '!(dstof x)'='!x'
AXIOM 'good(dstof x)'='good x'
AXIOM '!(src1of x)'='!x'
AXIOM 'good(src2of x)'='good x'
AXIOM '!(src2of x)'='!x'

opclassof LD = LCLASS
opclassof ST = SCLASS
opclassof ADD = ACLASS
opclassof SADD = ACLASS
opclassof ICOP = ACLASS
opclassof MOVE = ACLASS
opclassof CNE = ACLASS
opclassof JMP = JCLASS
opclassof JIT = JCLASS
opclassof JIF = JCLASS

|| We need a NO_OP, which we take to be ADD R0 0 R0, which adds 0 to R0.
noop :: data
AXIOM 'opcodeof noop'='ADD'
AXIOM 'indirect noop'='False'
AXIOM 'src1of noop'='R0'
AXIOM 'src2of noop'='DATUM n0'
AXIOM 'dstof noop'='DATUM n0'

|| This is the type of error flags. It's the part of the abstract state
|| which indicates whether an error has occurred.
error ::= NO_ERROR | JUMP_ERROR | STORE_ERROR

|| This function is used alot.
update f x y z = (x=z)->y; f z

reset_to_addr x = data_0

||clio: mod * off
||clio: add *
||clio: mod * on
```

6.2 The Axioms

```
AXIOM 'majority_of x x z'='x' , '!z'='True'
AXIOM 'majority_of x z x'='x' , '!z'='True'
AXIOM 'majority_of z x x'='x' , '!z'='True'
AXIOM "majority_commutes" 'majority_of x y z'='majority_of y z x'
AXIOM 'good data_0'='True'
AXIOM '!(msbof x)'='True' , '!x'='True'
AXIOM 'msbof bottom'='bottom'
AXIOM 'good x'='True' => 'good (inc_data x)'='True'
AXIOM 'good(add_data x y)'='True' , 'good x & good y'='True'
AXIOM "sregaddr flat" (x::sregaddr) '!!x'='!x'
AXIOM Proper_voterstate 'byzstep s in' , Proper_voterstate 's'
AXIOM "Proper_byzCayuga" Proper_ftcayuga 'byzCayugaStep s in' , Proper_ftcayuga 's'
AXIOM 'msbof (stwd1 st <<fr,x>> y)'='fr'
AXIOM 'next2msbof (stwd1 st x <<snd,y>>)'='snd'
AXIOM '!(stwd1 st x y)'='True' , '!!st & !!x &!!y'='True'
```

7 The Main Lemmas Proved

MainTheorem defines the main correctness property that we proved. It expresses the 12-cycle interactive consistency property for the entire system, which was described in detail in sections 5.2 and 9.1 of Volume 1. In the following, we describe some of the main lemmas that we proved in constructing a proof of MainTheorem.

MainTheorem states the interactive consistency property for the actual value (defined by the function actualPVT) that the processor sends to the voter. Since the processor sends the value to the voter over two cycles, we define one half of this value based on the contents of the special register PVT in the cycle in which ICOP is executed and the other half based on the contents of PVT in the following cycle. To ensure that the IC computation is performed on a consistent value of PVT, the programmer must make sure that the content of PVT will not change before the processor sends it to the voter. One way a programmer can ensure this is by not having a SADD instruction, which is the only instruction that can change the special register PVT, immediately following ICOP. AnotherMainTheorem formally states this condition. We have not proved this theorem, which should follow from the MainTheorem.

To decompose the proof of MainTheorem into smaller proofs, we divided the 12-cycle IC computation into a sequence of six *stages* of smaller duration. For each of these stages, we proved a *stage lemma* that defines the expected behavior for that stage. The different stages we used and their corresponding lemmas are enumerated below.

1. *Load stage* represents the first three cycles of IC computation in which the contents of the private register is moved from the processor to the voter. The lemma corresponding to this stage is `Load_correct_lemma`.

2. *Exchange stage* represents the next six cycles of the computation in which the voters exchange values between each other. This stage is further divided into a sequence of three 2-cycle computations, with one lemma for each of them: **`Xng1_correct_lemma`**, **`Xng2_correct_lemma`** and **`Xng3_correct_lemma`**.

3. *Compute stage* represents the single cycle in which majority computation is performed. The lemma corresponding to this stage is `Compute_correct_lemma`.

4. *Output stage* represents the last two cycle of the computation in which the results are written into the processor. The lemma corresponding to this stage is `Output_correct_lemma`.

We "chained" the stage lemmas together, in order, to prove the `MainTheorem`. We proved another set of lemmas, called *connect* lemmas, to assist us in the chaining process. That is, we used the connect lemmas to prove that the antecedent of a stage lemma implies the consequent of a succeeding stage lemma in the chain. The list given below shows the order in which we used the lemmas for proving the `MainTheorem`. In the following, `CL` is an abbreviation for "`Connect_lemma`," and $\rightarrow$ is used to indicate the chronological order in which two lemmas are proved when the lemmas appear within the same item in the list.

1. `Load_correct_lemma`
2. `CL_1_4` $\rightarrow$ (`CL_1_5`, `CL_1_3`) $\rightarrow$ (`CL_1_1`, `CL_1_2`)
3. `Xng1_correct_lemma`
4. (`CL_1_1`, `CL_1_2`) $\rightarrow$ `CL_3`
5. `Xng2_correct_lemma`
6. (`CL_1_1`, `CL_1_2`) $\rightarrow$ `CL_4`
7. `Xng_correct_lemma`
8. (`CL_1_1`, `CL_1_2`) $\rightarrow$ `CL_5`
9. `Compute_correct_lemma`.

10. (CL_1_1, CL_1_2) → CL_6

11. Output_correct_lemma.

```
||*************************************************************************
|| Now we specify the multi-cycle behavior of the ic net.
|| We only expect it to perform correctly under certain assumptions.
|| First, at the beginning of the multi-cycle computation, all the
|| non-faulty voters should be syncronized in their LDP1 state,
|| and be waiting for the go-signal.
|| And all the non-faulty processors should be about to give the go-signal.
|| And all the non-faulty processors should have their bytcounter at BYTE1.
|| Using the following predicates, we can assert this condition
|| on the voternet state, s, as:  Sync 'LDP1' 's' & All_go 's'

Sync 'cs' '<<ftc,vtr,inlist>>' :=
    ('faulty ONE' = 'False' => 'control (vtr ONE)'='cs')
    & ('faulty TWO' = 'False' => 'control (vtr TWO)'='cs')
    & ('faulty THREE' = 'False' => 'control (vtr THREE)'='cs')
    & ('faulty FOUR' = 'False' => 'control (vtr FOUR)'='cs')

vtr <<f,v,in>> = v
ftc <<f,v,in>> = f
inlist <<f,v,in>> = in
GO = <<True,True,True,True>>
go_signal <<ftc,vtr,int:rest>> index = vstartout (ftc index)

All_go 's' :=
    ('faulty ONE'='False' =>
        ('go_of (vtr s ONE)'='False'
         & 'go_signal s ONE'='GO' & 'bytecount (ftc s ONE)'='BYTE1'))
    & ('faulty TWO'='False' =>
        ('go_of (vtr s TWO)'='False'
         & 'go_signal s TWO'='GO' & 'bytecount (ftc s TWO)'='BYTE1'))
    & ('faulty THREE'='False' =>
        ('go_of (vtr s THREE)'='False'
         & 'go_signal s THREE'='GO' & 'bytecount (ftc s THREE)'='BYTE1'))
    & ('faulty FOUR'='False' =>
        ('go_of (vtr s FOUR)'='False'
         & 'go_signal s FOUR'='GO' & 'bytecount (ftc s FOUR)'='BYTE1'))

Bytecount_sync 'b' 's' :=
    ('faulty ONE'='False' => 'bytecount (ftc s ONE)'='b')
    & ('faulty TWO'='False' => 'bytecount (ftc s TWO)'='b')
    & ('faulty THREE'='False' => 'bytecount (ftc s THREE)'='b')
    & ('faulty FOUR'='False' => 'bytecount (ftc s FOUR)'='b')

|| So, the preconditions to the main correctness theorem are:

Preconditions 's' :=
    Proper_icnet 's' & Sync 'LDP1' 's' & All_go 's'

||*************************************************************************
|| Some intermediate results.

|| Specification of the load cycles.
|| This function defines the actual private value used by ICOP
actualPVT s index =
    half_and_half (newPVT2 (ftc s index) (first_interrupt s index))
                  (newPVT (ftc s index))

half_and_half hi lo = DATUM (Word (getbyte BYTE3 hi)(getbyte BYTE2 hi)
```

```
                                    (getbyte BYTE1 lo)(getbyte BYTE0 lo))

first_interrupt <<ftc,vtr,inlist>> index = select_int index (hd inlist)

load_correct 'before' 'after' 'index' :=
   ('faulty (succ index)'='False'
     => 'arrayof (vtr after index) 1 1' ='actualPVT before (succ index)')
   & ('faulty (succ2 index)'='False'
       => 'arrayof (vtr after index) 1 2' ='actualPVT before (succ2 index)')
   & ('faulty (succ3 index)'='False'
       => 'arrayof (vtr after index) 1 3' ='actualPVT before (succ3 index)')

Load_correct 'before' 'after' :=
    'faulty ONE'='False' => load_correct 'before' 'after' 'ONE'
    & 'faulty TWO'='False' => load_correct 'before' 'after' 'TWO'
    & 'faulty THREE'='False' => load_correct 'before' 'after' 'THREE'
    & 'faulty FOUR'='False' => load_correct 'before' 'after' 'FOUR'

Load = Iterate (#3) IcNetStep

Load_correct_proper 's' :=
    Load_correct 's' 'Load s' & Sync 'XNG11' 'Load s'
Load_correct_lemma :=
   Preconditions 's' => Load_correct_proper 's'

||**************************************************************************
|| Specification of the exchange cycles.
xng1_correct 's' 'index' :=
   ('faulty (succ index)'='False' =>
     'arrayof (vtr s index) 2 2'='arrayof (vtr s (succ index)) 1 1')
   & ('faulty (succ2 index)'='False' =>
       'arrayof (vtr s index) 3 3'='arrayof (vtr s (succ2 index)) 1 1')

xng2_correct 's' 'index' :=
   ('faulty (succ index)'='False' =>
      'arrayof (vtr s index) 2 3'='arrayof (vtr s (succ index)) 1 2')
   & ('faulty (succ3 index)'='False' =>
       'arrayof (vtr s index) 3 1'='arrayof (vtr s (succ3 index)) 1 2')

xng3_correct 's' 'index' :=
   ('faulty (succ2 index)'='False' =>
        'arrayof (vtr s index) 2 1'='arrayof (vtr s (succ2 index)) 1 3')
   & ('faulty (succ3 index)'='False' =>
        'arrayof (vtr s index) 3 2'='arrayof (vtr s (succ3 index)) 1 3')

Xng1_correct 's' :=
    'faulty ONE'='False' => xng1_correct 's' 'ONE'
    & 'faulty TWO'='False' => xng1_correct 's' 'TWO'
    & 'faulty THREE'='False' => xng1_correct 's' 'THREE'
    & 'faulty FOUR'='False' => xng1_correct 's' 'FOUR'

Xng2_correct 's' :=
    'faulty ONE'='False' => xng2_correct 's' 'ONE'
    & 'faulty TWO'='False' => xng2_correct 's' 'TWO'
    & 'faulty THREE'='False' => xng2_correct 's' 'THREE'
    & 'faulty FOUR'='False' => xng2_correct 's' 'FOUR'

Xng3_correct 's' :=
    'faulty ONE'='False' => xng3_correct 's' 'ONE'
    & 'faulty TWO'='False' => xng3_correct 's' 'TWO'
    & 'faulty THREE'='False' => xng3_correct 's' 'THREE'
    & 'faulty FOUR'='False' => xng3_correct 's' 'FOUR'

Xng = Iterate (#2) IcNetStep

Xng1_correct_lemma :=
```

```
    (Proper_icnet `s` & Sync `XNG11` `s`) =>
      (Xng1_correct `Xng s` & Sync `XNG21` `Xng s`)

Xng2_correct_lemma :=
    (Proper_icnet `s` & Sync `XNG21` `s` & Xng1_correct `s`) =>
      (Xng2_correct `Xng s` & Sync `XNG31` `Xng s` & Xng1_correct `Xng s`)

Xng3_correct_lemma :=
 (Proper_icnet `s` & Sync `XNG31` `s` & Xng1_correct `s` & Xng2_correct `s`) =>
 (Xng3_correct `Xng s` & Sync `CMPP` `Xng s`
  & Xng1_correct `Xng s` & Xng2_correct `Xng s`)

||**********************************************************************
|| The compute and output cycles:
|| An array is a function of type :: INDEX->INDEX->data
|| If A is an array and i is an index, the A i has type :: INDEX->data
|| so it represents a vector of four values, one for each index.
|| This representation makes it easy to rotate the vector, since the
|| rotated vector is obtained by composing with the function "succ".
|| Now we can say that an array A is consistent if, for all non-faulty
|| indicies, i and j, the vectors , A i and A j, are related by the appropriate
|| rotation, e.g. if j = succ2 i then (A i).succ2 equals (A j).


IndexConsistent `array` `index` :=
 (`faulty (succ index)`=`False`=>`(array index).succ`=`array (succ index)`)
 & (`faulty (succ2 index)`=`False`=>`(array index).succ2`=`array (succ2 index)`)
 & (`faulty (succ3 index)`=`False`=>`(array index).succ3`=`array (succ3 index)`)

Consistent `array` :=
 `faulty ONE`=`False`=> IndexConsistent `array` `ONE`
 & `faulty TWO`=`False`=> IndexConsistent `array` `TWO`
 & `faulty THREE`=`False`=> IndexConsistent `array` `THREE`
 & `faulty FOUR`=`False`=> IndexConsistent `array` `FOUR`


a_icvec init final index ONE = actualPVT init index
a_icvec init final index TWO = maj_val 1 final index
a_icvec init final index THREE = maj_val 2 final index
a_icvec init final index FOUR = maj_val 3 final index

maj_val i <<ftc,vtr,in>> index = get_maj_val i (maj_of (vtr index))

Compute = Iterate (#1) IcNetStep

Compute_correct_lemma :=
 (Proper_icnet `s` & Sync `CMPP` `s`
   & Proper_icnet `before` & Load_correct `before` `s`
   & Xng1_correct `s` & Xng2_correct `s` & Xng3_correct `s`)
  => (Consistent `a_icvec before (Compute s)`
      & Sync `OUT1` `Compute s`)

|| We want the final correctness condition to refer only to the
|| states of the processors, not the voters. So the icvec that
|| we state the condition on is:

icvec init final index ONE = actualPVT init index
icvec init final index TWO = get_special_register VT1 final index
icvec init final index THREE = get_special_register VT2 final index
icvec init final index FOUR = get_special_register VT3 final index

get_special_register saddr <<ftc,vtr,inlist>> index = sregof (ftc index) saddr

|| So to finish of the proof of correctness we need only show that
|| for non-faulty index, `icvec init (Output s) index`=`a_icvec init s index`
```

```
Output = Iterate (#2) IcNetStep

output_correct `s` `index` :=
    `icvec init (Output s) index`=`a_icvec init s index`

Output_correct `s` :=
    (`faulty ONE`=`False` => output_correct `s` `ONE`)
    & (`faulty TWO`=`False` => output_correct `s` `TWO`)
    & (`faulty THREE`=`False` => output_correct `s` `THREE`)
    & (`faulty FOUR`=`False` => output_correct `s` `FOUR`)

Output_correct_lemma :=
    (Proper_icnet `s` & Sync `OUT1` `s` & Bytecount_sync `BYTE1` `s`)
        => Output_correct `s`


Twelve_cycle = Output.Compute.Xng.Xng.Xng.Load

MainTheorem := Preconditions `s` => Consistent `icvec s (Twelve_cycle s)`

|| To chain these lemmas together we need:

Connect_lemma_1_1 :==
  (Proper_icnet `s` & Sync `cs` `s` & `cs=LDP1`=`False`)
     => Proper_icnet `IcNetStep s`
Connect_lemma_1_2 :==
  (Proper_icnet `s` & Sync `cs` `s` & `cs=LDP1`=`False`)
     => Sync `next_voterstate cs` `IcNetStep s`
Connect_lemma_1_3 :==
    Preconditions `s` => Sync `LDP2` `Iterate (#2) IcNetStep s`
Ready `s` :=
    (`faulty ONE`=`False` => `go_of (vtr s ONE)`=`True`)
    & (`faulty TWO`=`False` => `go_of (vtr s TWO)`=`True`)
    & (`faulty THREE`=`False` => `go_of (vtr s THREE)`=`True`)
    & (`faulty FOUR`=`False` => `go_of (vtr s FOUR)`=`True`)
    & Sync `LDP1` `s`

Connect_lemma_1_4 :==
    Preconditions `s` => (Proper_icnet `IcNetStep s` & Ready `IcNetStep s`)
Connect_lemma_1_5 :==
    (Proper_icnet `s` & Ready `s`)=> Proper_icnet `IcNetStep s`

Connect_lemma1 :=
  Connect_lemma_1_1 & Connect_lemma_1_2 & Connect_lemma_1_3 & Connect_lemma_1_4
Connect_lemma3 :=
     (Proper_icnet `s`  & Sync `XNG11` `s` & Load_correct `before` `s`) =>
                Load_correct `before` `Xng s`
Connect_lemma4 :=
     (Proper_icnet `s`  & Sync `XNG21` `s` & Load_correct `before` `s`) =>
                Load_correct `before` `Xng s`
Connect_lemma5 :=
     (Proper_icnet `s`  & Sync `XNG31` `s` & Load_correct `before` `s`) =>
                Load_correct `before` `Xng s`
Connect_lemma6 := Preconditions `s` =>
                  Bytecount_sync `BYTE1` `Compute(Xng(Xng(Xng(Load s))))`

||**********************************************************************
|| Here's another theorem that expresses a property useful from the point of
|| view of a programmer using ICOP.  It ensures that the system performs IC
|| computation on the contents of SREG[PVT] in the cycle in which ICOP was
|| executed, provided the instruction immediately following ICOP is not
|| SADD.  The following theorem, which has not been proved yet, can be proved
||  using the above MainTheorem as a lemma.
||**********************************************************************
|| After 12 cycles, each non-faulty processor should have a vector
|| of four values: the initial value of its PVT special register
```

```
|| and the final values in the special register VT1, VT2, and VT3.
|| Among the non-faulty processors, these vectors should be related
|| by a rotation.

anothericvec init final index ONE = get_special_register PVT init index
anothericvec init final index TWO = get_special_register VT1 final index
anothericvec init final index THREE = get_special_register VT2 final index
anothericvec init final index FOUR = get_special_register VT3 final index


|| Note that the expression "icvec init final index" is actually
|| a function on the type INDEX, and represents the vector of four values.
||*************************************************************************
ResultConsistent 's' := Consistent 'anothericvec s (Iterate #12 IcNetStep s)'

AnotherMainTheorem :=
    Preconditions 's' & Next_instrn_isnot_SADD 's' => ResultConsistent 's'

Array s = arrayof.(vtr s)
ItStep n = Iterate n IcNetStep
```

8 General Lemmas

```
'bytecount (FtCayugaStep s in)'='byte_inc(byte_inc(bytecount s))'

 'getbyte b (set_byte c w v)'= 'getbyte b w',  'b = c'= 'False'

'getbyte b (set_byte b w v)'='v' , 'good w & !v & !b'='True'

'DATUM (Word (getbyte BYTE3 x)(getbyte BYTE2 x)(getbyte BYTE1 x)
       (getbyte BYTE0 x))'='x' , 'good x'='True'

'!(msbof (getbyte b w))'='True' , '!b & good w'='True'

'(BYTE3 = x)|(BYTE2 = x)|(BYTE1= x)|(BYTE0=x)'='!x'

'!(getbyte b w)'='!b' , 'good w'='True'

'!!(data_to_addr x)'='True' , 'good x'='True'

'good(set_byte b w v)'='!b & !v' , 'good w'='True'

"good2definite" 'good x'='True' => '!!x'='True'

'!(byte_inc b)'='!b'

Proper_inlist 'a:b' => Proper_inlist 'b'

'!(byzCayugaStep s in)'='True' , Proper_ftcayuga 's'

'!(newcontr cs in irg)'='!cs' , '!in & !irg'='True'
```

References

[1] M. Bickford, C. Mills, and E.A. Schneider. "Clio: An Applicative Language-Based Verification System". Technical Report TR 89-13, ORA Corporation, 301A Harris B. Dates Drive, Ithaca, NY 14850, September 1989.

[2] R. Shostak M. Pease and L. Lamport. "Reaching Agreement in the Presence of Faults". *JACM*, 27(2):228–234, April 1980.

[3] Mandayam Srivas. "Bridging the Formal Methods Gap: A Computer-Aided Verification Tool for Hardware Designs". In *COMPCON91*, San Francisco, CA, February 25-28 1991.

[4] Mandayam Srivas and Mark Bickford. "Formal Verification of a Pipelined Microprocessor". *IEEE Software*, September 1990.

[5] Mandayam Srivas and Mark Bickford. "Verification of the FtCayuga Fault-Tolerant Microprocessor System Volume 1: A Case Study in Theorem Prover-Based Verification". Technical Report NASA Contractor Report 4381, NASA Langley Research Center, Hampton, VA 23665-5225, 1991. Authors' affiliation: ORA Corporation, 301A Dates Drive, Ithaca, NY 14850.

[6] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. "Formal Design and Verification of a Reliable Computing Platform for Real-Time Control Phase 1 Results". Technical Report NASA Technical Memorandum 102716, NASA, Langley Research Center, Hampton, Virginia 23665-5225, October 1990.

NASA

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-187574		

4. Title and Subtitle

Verification of the FtCayuga Fault-Tolerant Microprocessor System

Volume 2: Formal Specification and Correctness Theorems

5. Report Date

July 1991

6. Performing Organization Code

7. Author(s)

Mark Bickford and Mandayam Srivas

8. Performing Organization Report No.

10. Work Unit No.

505-64-10-05

9. Performing Organization Name and Address

ORA Corporation
301A Harris B. Dates Drive
Ithaca, NY 14850-1313

11. Contract or Grant No.

NAS1-18972

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

13. Type of Report and Period Covered

Contractor Report

14. Sponsoring Agency Code

15. Supplementary Notes

Technical Monitor: Ricky W. Butler, Langley Research Center
Task 1 Final Report
Volume 1 published as NASA CR-4381.

16. Abstract

This is the second volume of a two-volume report that presents a formal specification and verification of a property of a quadruplicately redundant fault-tolerant microprocessor system design. This volume gives a complete listing of the formal specification of the system and the correctness theorems that were proved. The system performs the task of attaining interactive consistency among the processors using a special instruction on the processors. The design is based on an algorithm proposed by Pease, Shostak and Lamport. The microprocessor used in the system is called FtCayuga, which was designed by extending another formally verified microprocessor MiniCayuga. The property verified ensures that an execution of the special instruction by the processors correctly accomplishes interactive consistency, provided certain preconditions hold, using a computer-aided hardware design verification tool, Spectool, and the theorem prover, Clio, both of which were developed at ORA. A major contribution of the work is the demonstration of a significant fault-tolerant hardware design that is mechanically verified by a theorem prover.

17. Key Words (Suggested by Author(s))		18. Distribution Statement
Fault Tolerance	Mechanical Theorem Proving	Unclassified - Unlimited
Formal Verification		
Byzantine Agreement		Subject Category 62
Hardware Verification		

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	74	A04